

Федеральное государственное автономное образовательное  
учреждение высшего образования  
"Национальный исследовательский университет  
"Высшая школа экономики"

Факультет МИЭМ

Департамент компьютерной инженерии

направления подготовки 09.03.01 "Информатика и вычислительная техника"  
уровень бакалавр

## **ПРОГРАММНАЯ МОДЕЛЬ ПРОЦЕССОРА И БАЗОВЫЕ ИНСТРУКЦИИ ПРОЦЕССОРОВ INTEL И MIPS**

Учебно-методическое пособие по дисциплине  
«Вычислительные системы и компьютерные сети»

Составитель:

доцент, кандидат технических наук,  
Е. М. Иванова,  
emivanova@hse.ru

Москва 2019

Составитель к.т.н., доц. Е. М. Иванова

Учебное пособие является составной частью методического обеспечения по дисциплине «Вычислительные системы и компьютерные сети», изучаемой студентами в рамках направления 09.03.01 – «Информатика и вычислительная техника».

Основным содержанием является изучение архитектуры компьютера на примере 32-разрядных процессоров Intel и MIPS, их системы команд, последовательности обработки информации при вычислениях с целыми, дробными, символьными данными и числами в двоично-десятичном представлении, векторами и работой со стеком. В процессе изучения для моделирования последовательности исполнения процессорами машинного кода студентам предлагается воспользоваться программными пакетами MASM32, OllyDbg и MARS.

## ЦЕЛЬ И СОДЕРЖАНИЕ УЧЕБНОГО ПОСОБИЯ

**Целью данного пособия** является закрепление теоретических знаний по разделу «архитектура набора команд» компьютера и получение практических навыков в программировании на Ассемблере для двух различных процессоров: Intel и MIPS.

### АННОТАЦИЯ

Для лучшего понимания организации работы процессора следует изучить его программную модель и систему команд. Студентам предлагается сравнить два класса процессоров CISC и RISC, программные модели которых отличаются наиболее сильно. В пособии рассмотрены вопросы выполнения операций в этих процессорах с разными типами данных. Предложены варианты практических задач, решение которых позволит лучше понять принципы выполнения базовых операций в процессоре. Студентам предлагается написать, отладить и смоделировать исполнения различных программ.

Практические задания предназначены для помощи студентам в освоении архитектуры вычислительных систем через практические примеры программирования на Ассемблере Intel и MIPS. Дополнительно предлагается ознакомиться с пакетом MASM, OllyDbg и MARS, их основных функций и возможностей.

## СОДЕРЖАНИЕ

<b>ГЛАВА 1. Программирование в среде MASM и OLLYDBG. ПР№1.....</b>	<b>7</b>
1. ТЕОРИЯ.....	7
1.1. Машинный язык и Ассемблер .....	7
1.2. Последовательность написания и отладки программ на Ассемблере .....	7
1.3. Программный пакет MASM32. Установка и настройка.....	10
1.4. Особенности программирования под Windows .....	11
1.5. Написание программы на Ассемблере .....	12
1.6. Первый пример Windows-программы .....	12
2. ПРАКТИЧЕСКИЕ ЗАДАНИЯ .....	14
2.1. Задача № 1 .....	14
2.2. Порядок выполнения Задания № 1 .....	14
2.3. Подготовка к отладке.....	16
2.4. Программный пакет OllyDbg .....	16
2.5. Форматы данных в Ассемблере Intel .....	28
2.6. Пример программы .....	29
2.7. Структура команды.....	33
3. ЗАДАНИЕ .....	35
3.1. Варианты заданий.....	36
4. КОНТРОЛЬНЫЕ ВОПРОСЫ .....	38
5. СПОСОБ ОЦЕНИВАНИЯ.....	40
6. СПИСОК ЛИТЕРАТУРЫ.....	40
<b>ГЛАВА 2. Программирование в среде MARS. ПР №2.....</b>	<b>41</b>
1. ТЕОРИЯ.....	41
1.1. Машинный язык и Ассемблер .....	41
1.2. Последовательность написания и отладки программ на Ассемблере.....	41
1.3. Программный пакет MARS. Установка и настройка .....	43
1.4. Написание программы на Ассемблере .....	43
1.5. Первый пример программы .....	43
1.6. Задание № 1 .....	45
1.7. Порядок выполнения Задания № 1 .....	45
1.8. Опции отладки.....	48

1.9.Пример программы .....	50
1.10.Структура команды.....	52
2. ЗАДАНИЕ .....	57
3. ВАРИАНТЫ ЗАДАНИЙ .....	58
4. КОНТРОЛЬНЫЕ ВОПРОСЫ .....	60
5. СПОСОБ ОЦЕНИВАНИЯ.....	61
6. СПИСОК ЛИТЕРАТУРЫ.....	61
<b>ГЛАВА 3. Адресация элементов массива, организация цикла. ПР №3 .....</b>	<b>62</b>
1. РАБОТА С МАССИВАМИ В INTEL.....	62
1.1.Пример обращения к элементу одномерного массива.....	62
1.2.Организация цикла for .....	63
1.3.Организация циклов while.....	65
1.4.Использование двумерных массивов (матриц).....	65
1.5.Пример обращения к элементу двумерного массива .....	66
1.6.Пример программы подсчёта количества нулевых элементов в двумерном массиве .....	67
1.7.Примеры различных режимов адресации операндов и команд .....	68
1.8.API-функция wsprintf.....	69
2 РАБОТА С МАССИВАМИ В MIPS.....	72
2.1. Обращение к элементу одномерного массива.....	72
2.2. Организация цикла (for, while).....	73
2.3. Использование двумерных массивов (матриц) .....	75
3 ЗАДАНИЕ .....	76
4 ВАРИАНТЫ ЗАДАНИЙ .....	76
5 КОНТРОЛЬНЫЕ ВОПРОСЫ .....	78
6 СПИСОК ЛИТЕРАТУРЫ.....	79
<b>ГЛАВА 4. Работа с действительными числами и регистрами FPU. ПР №4 .....</b>	<b>80</b>
1. ТЕОРИЯ.....	80
1.1. Формат чисел с плавающей запятой (ЧПЗ или FP) .....	80
1.2. Работа с ЧПЗ в процессоре Intel.....	85
1.3. Вывод ЧПЗ в окно .....	94
1.4. Работа с ЧПЗ в процессоре MIPS.....	95
2. ЗАДАНИЕ .....	99
3. ВАРИАНТЫ ЗАДАНИЙ .....	100

_Тос245823574.КОНТРОЛЬНЫЕ	ВОПРОСЫ
.....	101
5. СПИСОК ЛИТЕРАТУРЫ.....	102
<b>ГЛАВА 5. Специальные типы данных: векторные операции, двоично-десятичная арифметика. ПР №5</b> .....	103
1. ТЕОРИЯ.....	103
1.1. Десятичная арифметика.....	103
1.2. Программирование на Ассемблере Intel .....	105
1.3. Арифметические операции над BCD-числами.....	106
1.4. Сложение неупакованных BCD-чисел .....	107
1.5. Вычитание неупакованных BCD-чисел .....	108
1.6. Умножение неупакованных BCD-чисел .....	111
1.7. Деление неупакованных BCD-чисел .....	112
1.9. Арифметические действия над упакованными BCD-числами.....	113
1.10. Сложение упакованных BCD-чисел.....	113
1.11. Вычитание упакованных BCD-чисел.....	115
1.12. Отрицательные BCD-числа.....	117
1.13. Многобайтовые BCD-числа.....	117
1.14. Вывод BCD-числа .....	118
2. ВЕКТОРНЫЕ ТИПЫ ДАННЫХ .....	121
2.1. Программирование под Intel .....	121
2.1. Программирование под MIPS .....	125
3. ЗАДАНИЕ .....	126
4. Контрольные вопросы .....	128
5. СПИСОК ЛИТЕРАТУРЫ.....	129
<b>ГЛАВА 6. стек. Вызов процедур. Передача параметров. ПР №6</b> .....	130
1. ТЕОРИЯ.....	130
1.1. Программирование на Ассемблере Intel.....	130
1.2. Программирование на Ассемблере MIPS .....	135
2. ЗАДАНИЕ .....	139
3. КОНТРОЛЬНЫЕ ВОПРОСЫ .....	139
4. СПИСОК ЛИТЕРАТУРЫ.....	139

# ГЛАВА 1. ПРОГРАММИРОВАНИЕ В СРЕДЕ MASM И OLLYDBG.

## Практическая работа №1

### 1. ТЕОРИЯ

#### 1.1. Машинный язык и Ассемблер

Самый низкий уровень программирования – машинный язык (последовательность двоичных кодов машинных команд и операндов). Писать программы с операторами в виде двоичных чисел очень сложно, поэтому во всех машинах предусмотрен язык ассемблера – символическое представление архитектуры набора команд, в котором двоичные числа заменены именами команд (мнемониками, наподобие ADD, SUB и MUL) и именами операндов (символьным обозначением ячеек памяти – VAR, string1, регистров – AX, DL, адресов – меток).

В качестве основы для изучения выбран 32-разрядный процессор Intel. Программы, написанные для этих процессоров, можно выполнять на большинстве современных систем, что позволяет читателю воочию наблюдать результаты своей работы.

Для того чтобы хорошо разбираться во внутреннем строении компьютера и любой вычислительной системы и уметь её программировать нужно хорошо освоить уровень архитектуры набора команд, который представляет собой совокупность регистров, команд и других элементов, доступных программистам, пишущим на языках низкого уровня (системных программистов). Для первого знакомства с программированием под Windows рассмотрим простейшую программу на Ассемблере. Для написания, транслирования и запуска будем использовать пакет MASM32.

#### 1.2. Последовательность написания и отладки программ на Ассемблере

Сначала создаётся файл, содержащий текст программы на Ассемблере. Этот файл должен иметь расширение '.asm'. Когда исходный текст набран и сохранён, его можно компилировать (переводить на машинный язык). Перевод программы,

написанной на языке Ассемблера (файлы с расширением ‘.asm’), на машинный язык двоичных кодов (файлы с расширением ‘.exe’) проходит в два этапа.

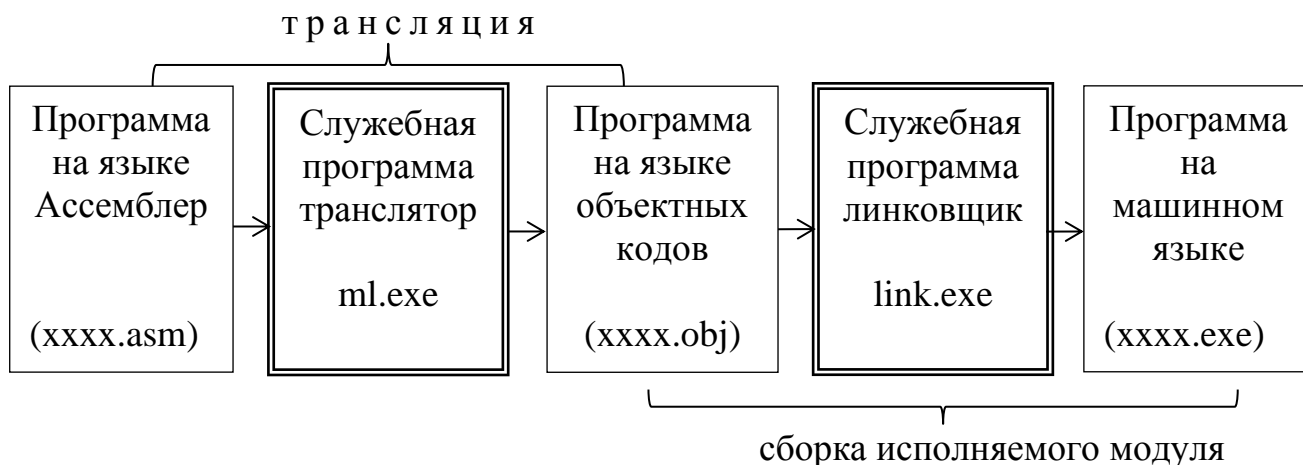
1. Исходный текст преобразуется в промежуточный файл – объектный файл (с расширением obj). Это делает транслятор (в пакете MASM он называется ml.exe).

2. Затем создаётся готовый исполняемый модуль – программа (запускаемый файл с расширением exe для DOS, Windows...). Это делает линковщик (в пакете MASM он называется link.exe).

Для того чтобы создавать программы, нужно иметь текстовый редактор и компилятор, который состоит из двух основных частей - транслятор и линковщик. Всё остальное не обязательно, однако реальные программы Win32 используют внешние функции, стандартные константы и переменные, ресурсы и много другое. Всё это требует дополнительных файлов, которые и включает пакет MASM32. Это очень распространённый пакет, собранный Стивеном Хатчисоном. Важно понять, что MASM32 вовсе не компилятор, а сборник для программирования под Win32, в который входит среди прочего 32-битный компилятор MASM. Эти файлы можно скачать, например, с сайта <http://www.masm32.com/>.







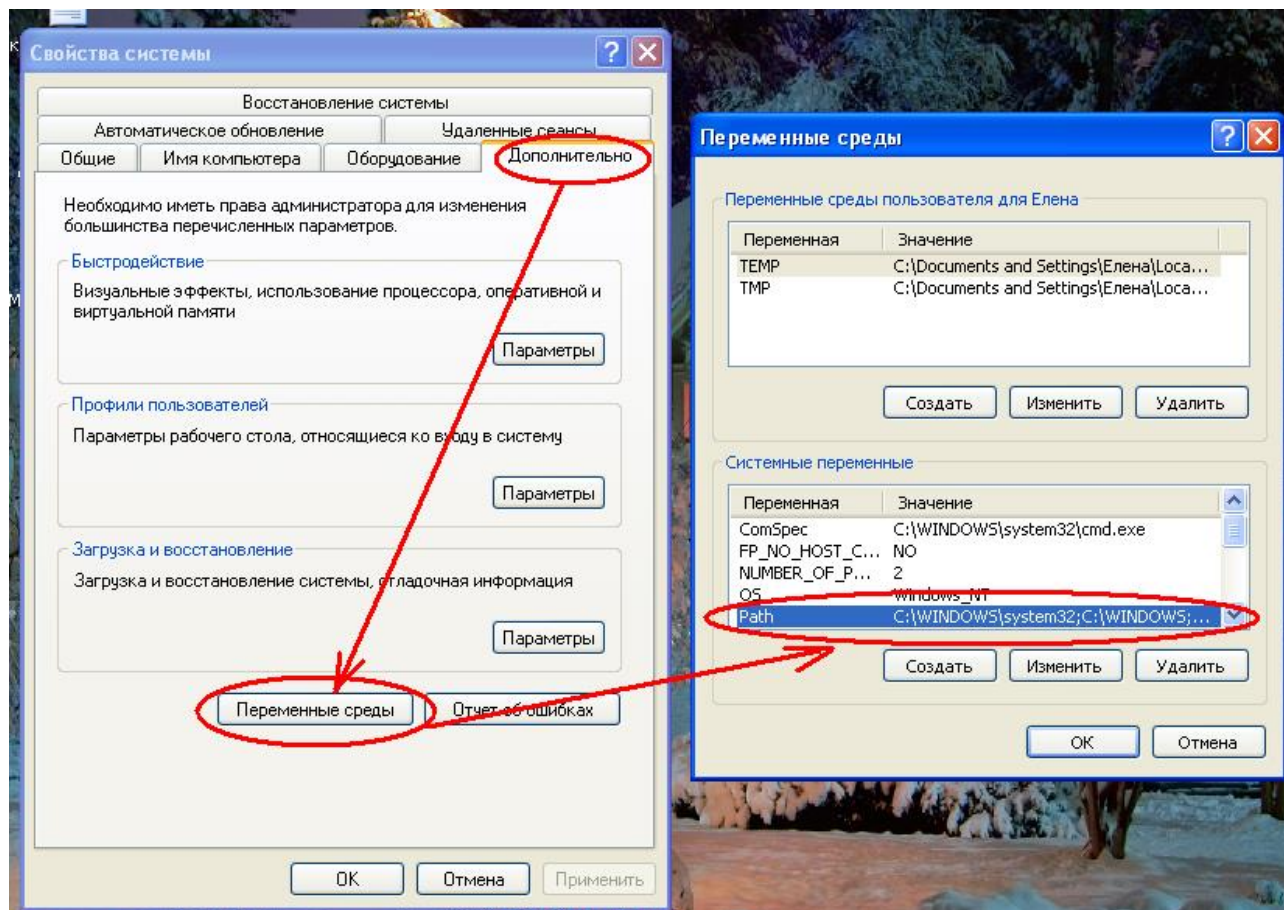
Программа в машинных кодах готова непосредственно к выполнению в реальной аппаратной среде. Однако, в этом случае, обнаружение ошибок усложняется. Чтобы упростить задачу программиста, двоичную программу можно запустить не в реальной аппаратной среде, а в отладчике/симуляторе, который имеет ряд возможностей, в том числе и работать в режиме интерпретатора.



В последнем случае каждый отдельный момент времени выполняет только одну команду (интерпретирует) и выводит детальный отчет о своих действиях. В этом случае отлаживать программы становится значительно проще. В среде отладчика программы выполняются значительно медленнее, чем в реальных условиях. Предлагается использовать достаточно наглядный отладчик – пакет OllyDbg.

### 1.3. Программный пакет MASM32. Установка и настройка

Перед началом работы вам нужно скопировать установочный файл для пакета MASM (из архива файлов системы LMS), установить этот пакет на диск вашего компьютера и выполнить небольшую настройку.



1. Перед установкой **временно** отключите чрезмерно бдительные антивирусы (Касперского в первую очередь).

2. Для простоты надо устанавливать пакет на тот же диск, где будут находиться и сами программы (исходные файлы с расширением .asm).

3. Требуется прописать в системе путь к файлам компилятора. В Win7 это можно сделать так. В меню "пуск" правый клик на "Компьютер" → Свойства → Дополнительные параметры системы → закладка "Дополнительно" → Переменные среды. Здесь в списке системных переменных нужно изменить значение переменной "PATH". Дописать в конце строки:

`;F:\MASM32\bin`

(не пропустите точку с запятой и вместо "F" укажите букву диска, на котором установлен MASM32).

В других версиях ОС операция выполняется по аналогии. Например, в Windows 10 можно добраться до настройки PATH так же.

Этот компьютер → Свойства → Дополнительные параметры системы → Дополнительно → Переменные среды (или вызовом «Изменение переменных среды текущего пользователя» в результатах поиска).

В окошке «Переменные среды» в блоке «Переменные среды пользователя %USERNAME%» находим строку PATH, выделяем кликом, ждем кнопку «Изменить...» и в конце строки «Значение переменной» вписываем точку с запятой и путь к папке

;F:\MASM32\bin

(на самом деле, важен путь к запускаемым файлам пакета: link.exe, ml.exe которые должны лежать в папке bin).

#### 1.4. Особенности программирования под Windows

После загрузки ОС Win32 происходит следующее.

Каждая программа загружается в собственное изолированное виртуальное адресное пространство (Виртуальное Пространство Процесса - ВПП).

Адреса в таком ВПП могут быть от 00000000 до FFFFFFFF (4Gb), будем называть их виртуальными.

Программа, кроме собственных вычислений, практически ничего самостоятельно сделать не может. Ей приходится запрашивать ОС вывести что-либо или предоставить в её ВПП. Для этого она вызывает нужные API-функции<sup>1</sup> (от англ. *application programming interface*) - интерфейс программирования приложений (иногда интерфейс прикладного программирования).

Файлы с машинным кодом и данными у Win32-программ устроены в соответствии с PE-форматом<sup>2</sup>, который довольно сильно связан с особенностями защищённого режима конкретного процессора. Исполняемые файлы PE-

---

<sup>1</sup> — набор готовых классов, процедур, функций, структур и констант, предоставляемых для удобства использования во всевозможных приложениях.

<sup>2</sup> — **PE** (*Portable Executable* - портируемый exe-формат).

формата делятся на секции. Должна быть минимум одна секция (секция кода), но файл может иметь и другие секции для разных целей (данные, ресурсы, служебные секции и т.д.).

### 1.5. Написание программы на Ассемблере

Для написания программы можно воспользоваться любым текстовым редактором.

- MsWord (при установленном фильтре простого текста),
- RadAsm (среда для разработки приложений),
- Quick Editor пакета MASM,
- блокнот AkelPad...

Однако лучше не использовать Word, т.к. он добавляет символы форматирования. Самое простое – воспользоваться стандартной программой «Блокнот». В пакете MASM имеется свой встроенный редактор - Quick Editor.

Программа должна быть сохранена с расширением .asm и помещена в стандартную папку «examples» пакета.

### 1.6. Первый пример Windows-программы

Ниже рассмотрен пример небольшой программы **primer.asm** для Win32, написанной под MASM. Сразу поясним, что фактически в этой программе нет ни одной команды на собственно Ассемблере. В этом вы можете убедиться, сравнив все встретившиеся слова со списком команд из [1] или [2].

Для удобства написания и отладки в asm-программах кроме команд Ассемблера используются ещё так называемые Директивы (операторы конкретного программного пакета, в нашем случае пакета MASM32) и вызовы API-функций ОС Win32.

С помощью директив удобно описывать и задавать переменные, массивы, строки, вызывать API-функции. Хотя любая директива может быть заменена на Ассемблерный код, писать каждый раз такой код заново не надо, он будет автоматически подставляться при компиляции или использоваться непосредственно при выполнении программы.

```

.386                ;директива использования системы команд процессора Intel.386
.model flat, stdcall ;директива использования сплошной модели ОП с передачей аргументов в функции через стек
                    ;с последующим возвращением вершины стека в то состояние, которое было до передачи аргументов
option casemap :none ;установка чувствительности к ПРОПИСНЫМ или строчным символам (STR≠str)

##### <----- просто для красоты
include \masm32\include\windows.inc      ;
include \masm32\include\user32.inc       ;
include \masm32\include\kernel32.inc     ;директивы подключения файлов и библиотек для вызова API-функций
;
includelib \masm32\lib\user32.lib         ;
includelib \masm32\lib\kernel32.lib      ;

#####
.data                ;директива объявления секции данных
;строковая переменная для вывода текста в названии окна (заканчивается нулевым байтом)
str1 db "Это ваша первая программа для Win32",0
;строковая переменная для вывода сообщения в окне
str2 db "Assembler для Windows – это сказка!",0

#####
.code               ;директива объявления секции кода
start:              ;метка – точка входа в программу – адрес первой исполняемой команды, который будет помещён в EIP
                    ;при запуске программы. При переходе по метке компилятор подставляет соответствующий ей адрес
                    invoke MessageBox, NULL, addr str2, addr str1, MB_OK
;вызов API-функции MessageBox – вывода на экран окна, и передача параметров: названия окна – str1 и сообщения – str2
                    invoke ExitProcess, NULL ;вызов завершающей функции с параметром NULL
end start           ;окончание программы

```

Программа **primer.asm** – это каркас любой вашей будущей Windows-программы, далее можно вписывать туда только полезный код.

В ней всё, что располагается справа после точки с запятой является комментарием и не транслируется в машинный код.

Более подробно описание работы пакета MASM можно найти на сайте «Дневники чайника» [3].

## 2. ПРАКТИЧЕСКИЕ ЗАДАНИЯ

### 2.1. Задача № 1

Научиться пользоваться пакетом MASM.

Создать программу **primer.asm** в редакторе Quick Editor пакета MASM.

Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторную компиляцию и запуск).

Внести в программу небольшие изменения (название окна, выводимое сообщение, добавить вывод ещё одного окна ...) и запустить новую изменённую программу.

### 2.2. Порядок выполнения Задания № 1

1. Откройте редактор исходных файлов "Quick Editor" (икона "MASM32 Editor" на рабочем столе или файл qeditor.exe в папке masm32).

2. Наберите в нём текст программы или скопируйте из данной методички. Можно несколько строк набрать, а остальное скопировать, используя стандартную опцию меню "Edit/Paste". Лишние строки комментариев можно не набивать.

3. После всех изменений обязательно нужно сохранить пример опцией меню «File/Save as» и обязательно на тот диск, где установлен MASM32 (например, в созданную вами папку «C:\masm32\examples\new»).

4. Для самой компиляции нужно выполнить пункт меню Project > Build All. При этом должно появиться консольное окошко с отчётом компилятора (если консоли нет, значит, по какой-то причине компиляции не было). Это действие

вызовет последовательно работу двух уже названных программ: транслятора ml.exe и линковщика link.exe. В текущей папке «C:\masm32\examples\new» с файлом **primer.asm** будут созданы два новых файла **primer.obj** и **primer.exe**. В их наличие можно убедиться, проверив содержимое папки **new**.

5. Запустите программу-пример можно из того же Quick Editor'a кнопкой "Run File" (пиктограмма в виде машинки) или пунктом меню Project > Run Program.

*Если вы планируете работать с компилятором на разных дисках, то самый грубый, но быстрый и универсальный, способ решить проблему компиляции – скопировать папку MASM32 в корень всех дисков, где будет проходить компиляция программ.*

Если при компиляции были обнаружены ошибки, об этом будет сообщение в окне командной строки с указанием номера строки или типа ошибки. Исправьте ошибки и запустите компиляцию повторно.

6. Внесите изменения в текст программы. После внесения изменений в исходный файл **primer.asm** перед его повторной компиляцией обязательно сохраните эти изменения в текущий файл опцией "Save" или в новый файл опцией "Save as". И запустите изменённую программу.

Данная программа достаточно проста и не требует отладки. Но для любых более сложных заданий для облегчения написания и отладки программ на Ассемблере требуется применить какой-либо отладочный пакет, позволяющий на каждом шаге выполнения программы отслеживаться изменения в регистрах процессора, переменных в памяти, менять по ходу выполнения команды программы, операнды, содержимое ячеек памяти, устанавливать точки останова и многое другое.

Рассмотрим, например, пакет OllyDbg<sup>3</sup> (Olly Debugger). Для написания, транслирования и запуска будем по-прежнему использовать пакет MASM32.

---

<sup>3</sup> *OllyDbg 1.10 32-разрядный отладчик уровня Ассемблера под ОС Windows. OllyDbg 2.01 32-разрядный отладчик уровня Ассемблера, адаптированный под 64-разрядный процессор и под ОС Windows.*

Обратите внимание на разрядность вашего компьютера. Если он 32-разрядный то подойдёт версия 1.1, если он 64-разрядный то требуется версия не ниже 2.01.

### **2.3. Подготовка к отладке**

Выполним уже знакомые нам действия.

Создадим текст исходной программы (см. задание № 2) в любом редакторе (например, в Quick Editor пакета MASM).

Сохраним текст в файле с расширением *.asm* (например, *prakt2.asm*) в любую папку на диске с пакетом MASM. Лучше для единообразия создать собственную папку в директории “examples”.

Запустим компиляцию через опцию меню Project > Build All.

Убедимся, что в тексте программы нет ошибок, и компиляция прошла успешно (в вашей папке появились два файла (*prakt2.obj* и *prakt2.exe*). если ошибки имеются, то нужно их исправить и повторить действия 2,3,4.

Запустим программу на исполнение опцией меню Project > Run Program или кнопкой "Run File" (пиктограмма в виде машинки).

Протестируем программу на различных наборах исходных данных. Если всё работает правильно, то выберем любой вариант исходного тестового набора и для него создадим исполняемый файл (*prakt2.exe*). В случае неверного функционирования программы для отладки и поиска логических и алгоритмических ошибок будем использовать ошибочную версию *prakt2.exe*. Следует отметить, что синтаксические ошибки уже были выявлены на этапе компиляции.

Запустим отладчик OllyDbg.

### **2.4. Программный пакет OllyDbg**

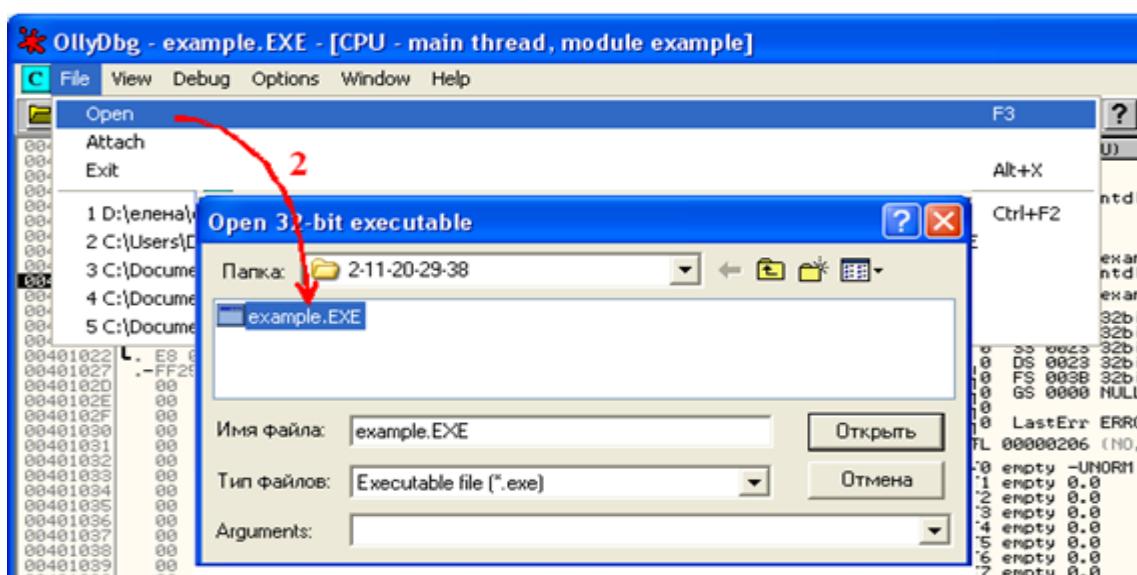
OllyDbg представляет собой 32-разрядный отладчик для анализа кода на уровне Ассемблера с интуитивно понятным интерфейсом. Работает под Windows 95, 98, ME, NT или 2000, XP (тестирование не на 100%), на любом компьютере



класса не ниже Pentium, но для удобной отладки требуется как минимум 300-МГц процессор. Если вы собираетесь использовать расширенные возможности (трассировка), рекомендуется 128 или более МВ оперативной памяти. Работа с ним достаточно подробно описана в литературе, например [4].

Сначала распакуем архив в папку на том же диске, где расположен и MASM. В созданной папке нужно запустить исполняемый файл OLLYDBG.exe. Можно для удобства создать ярлык на рабочем столе. Установки не требуется, можно запустить приложение (файл «OLLYDBG.EXE») из любой папки.

После запуска открывается окно отладчика. Стандартным способом через опцию меню «File/Open» находим нужный файл (*prakt2.exe*) и открываем его.

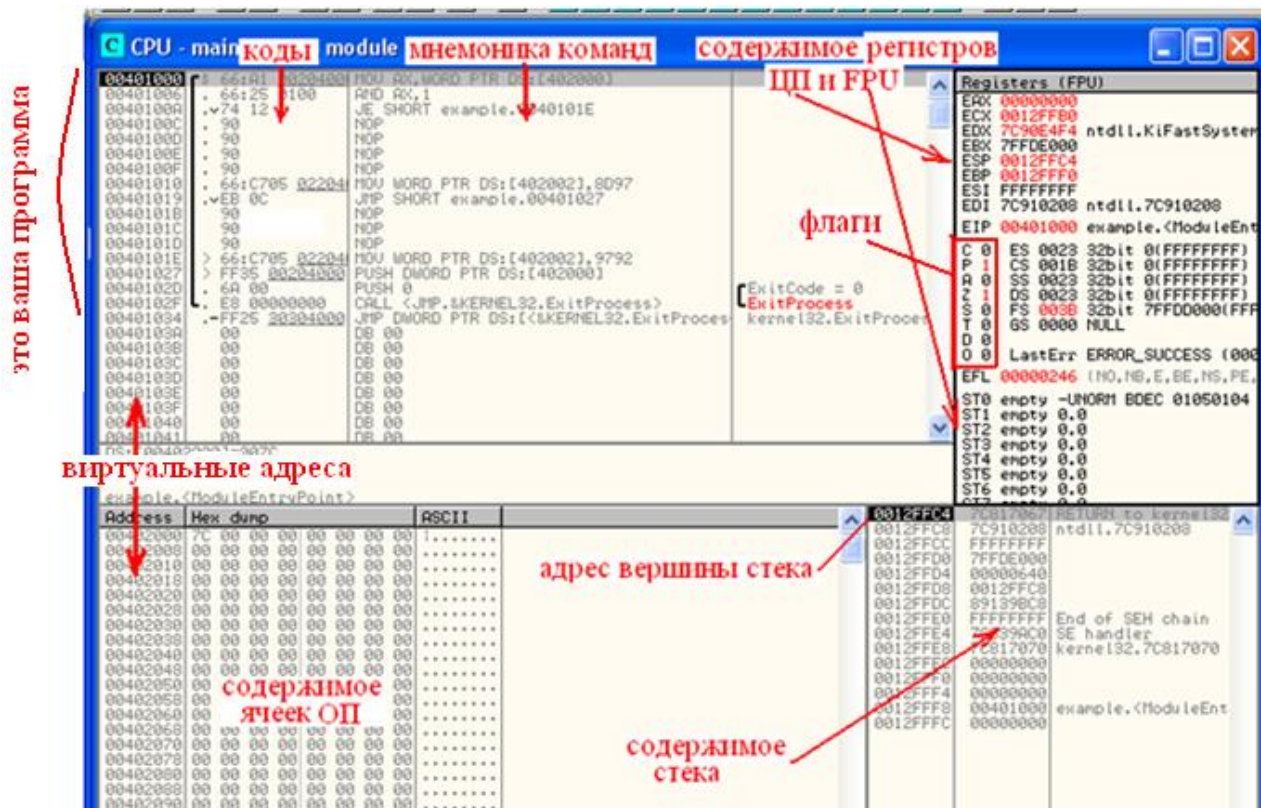



В поле отладчика появится текст кода и параллельно текст на ассемблере.

Здесь в отдельных окошках можно увидеть

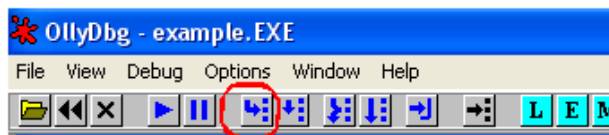
- текст фрагмента программы в машинных кодах и на ассемблере (мнемонику команд),
- адреса каждой команды и первой ячейки памяти в отображаемой строке,
- коды команд и коды данных в ячейках памяти в шестнадцатеричной системе счисления (по желанию можно выбрать другое представление, см. [4]),
- содержимое регистров процессора CPU и сопроцессора FPU,
- содержимое и адреса ячеек стека.

Размеры и границы окон могут быть изменены привычным для Windows способом.

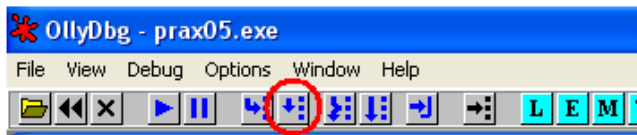


Далее вы должны выполнять программу по шагам. За один шаг выполняется одна машинная команда. Для этого требуется либо последовательно нажимать клавишу F7, либо иконку .

Также можно пользоваться кнопкой F8. Отличие их в том, что в последнем случае программа выполняется пошагово без захода в подпрограммы (команда CALL сразу дает результат работы подпрограммы).





F7



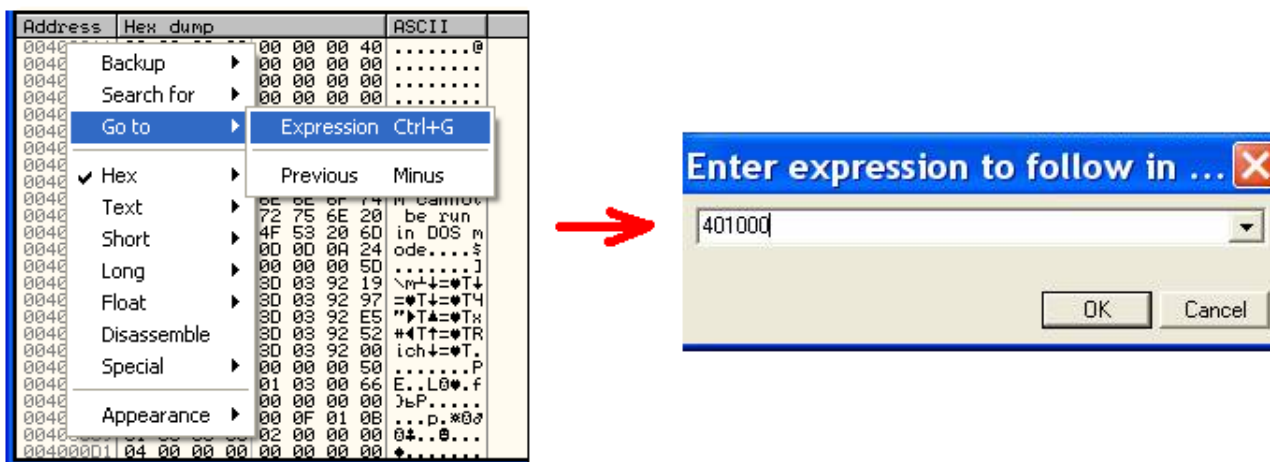
F8

Для простой первой программы будем пользоваться этим последним способом отладки - F8. Назначение остальных кнопок можно посмотреть в Help или в [4]. Кратко укажем, что

- кнопка F2 добавляет/убирает обычную точку останова на отмеченной строке.

- кнопка F9 или иконка  запускает программу, которая будет выполняться до тех пор, пока не встретит точку останова или в отсутствии последней до конца;
- кнопка F12 или иконка  временно прекращает выполнение программы.

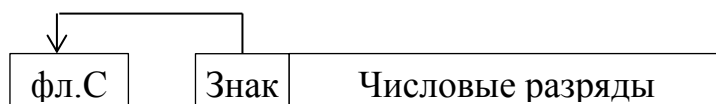
Для перемещения к любому адресу в окне кода или окне данных достаточно кликнуть в поле этого окна правой кнопкой мыши и выбрать в выпадающем меню опцию “Go to/Expression”, а затем в появившемся окошке набрать нужный адрес.



После каждого шага нужно отслеживать, как повлияла выполненная команда на состояние процессора и памяти (содержимое регистров, флагов, указателя адреса следующей команды EIP, регистра адреса вершины стека ESP, содержимое ячеек памяти по определенному адресу...).

Напомним значение флагов (разрядов регистра EFL):

**флаг C** (переноса) – равен «1», если был перенос из старшего разряда числа при последней арифметической операции (ADD, SBB) или операции сдвига (ROL);



**флаг P** (чётности) – равен «1», если в двоичном представлении младшего байта результата чётное количество единиц;

**флаг A** (дополнительного переноса) – равен «1», если был перенос из 3 в 4 разряд результата (используется для двоично-десятичной арифметики);

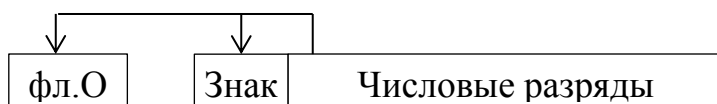
**флаг Z** (нуля) – равен «1», если результатом операции является ноль;

**флаг S** (знака) – равен старшему разряду (знаку) результата;

**флаг Т** (трассировки) – используется для особого режима отладки;

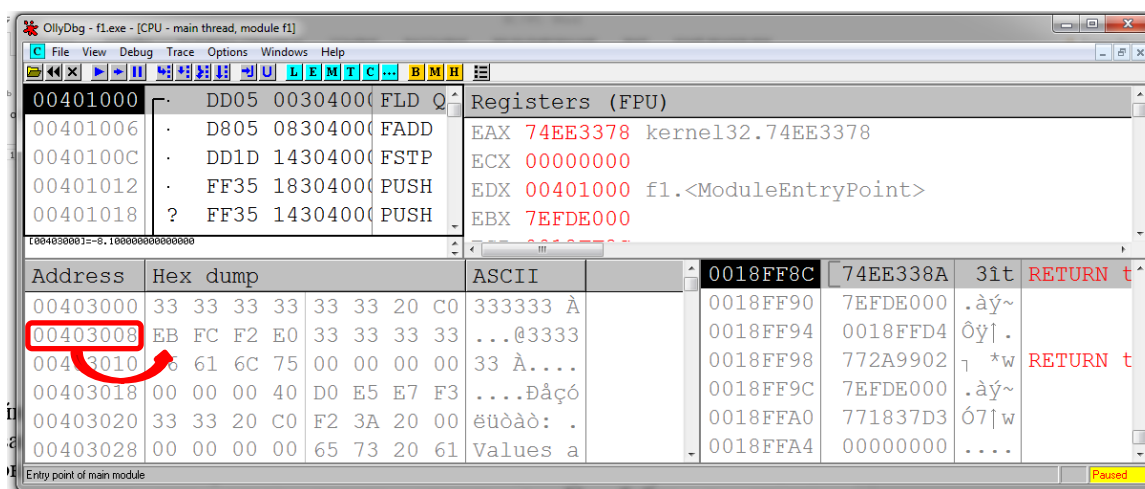
**флаг D** (направления просмотра строк) – равен «0» при прямом направлении от начала к концу, «1» – от конца к началу;

**флаг O** (переполнения) – показывает, был ли перенос из старшего числового разряда в знаковый разряд при операции сложения или сдвига.



Заметьте, что в языке Ассемблера Intel не все команды влияют на флаги, а только команды обработки данных. Команды передачи данных и передачи управления на флаги не влияют.

Для экономии места все ячейки ОП расположены как в таблице – построчно. Размер строки – 8 байт (можно установить другое, см. [4]). В начале строки указан адрес первого в строке байта. Адреса остальных байтов при необходимости можно рассчитать самостоятельно, используя шестнадцатеричную арифметику.



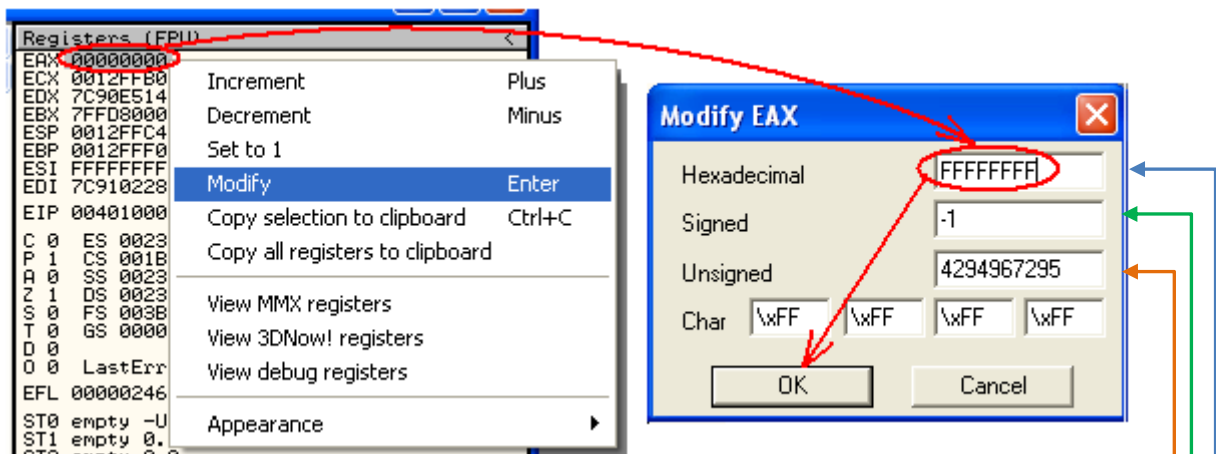
Например, на рис. выше выделен красным адрес 00403008 – это номер крайнего слева – нулевого байта второй строки окна HEX DUMP (содержимое указанного байта EB), Адрес первого байта (FC) рассчитывается как адрес начала строки + № байта в строке (00403008+1=00403009). Адрес второго байта (F2) рассчитывается как 00403008+2=0040300A... Адрес последнего седьмого байта (33) рассчитывается как 00403008+7=0040300F.

**Address Hex dump**

00403008 FE FF FF FF| 04 00 00 00|  
 00403008 00403008+1 00403008+2 00403008+7=0040300F

### Рассчитанные адреса байтов строки

Содержимое любого регистра и любой ячейки памяти может быть изменено для проверки (в текст исходной программы на Ассемблере *prakt2.asm* эти изменения не попадут). Изменим, например, значение **EAX** на то, которое нужно нам, в данный момент для проверки. Для этого кликнем правой кнопкой мыши на поле выбранного регистра (или дважды кликнем левой кнопкой мыши на самом значении). В выпадающем меню выберем опцию «Modify», а затем в открывшемся окне модификации просто введём новое значение и нажмём кнопку «ОК».



Новое значение регистра можно задавать в любом удобном для вас формате:

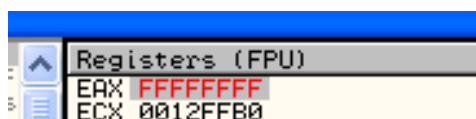
шестнадцатеричном (**Hexadecimal**),

десятичном со знаком (**Signed**),

десятичном без знака (**Unsigned**).

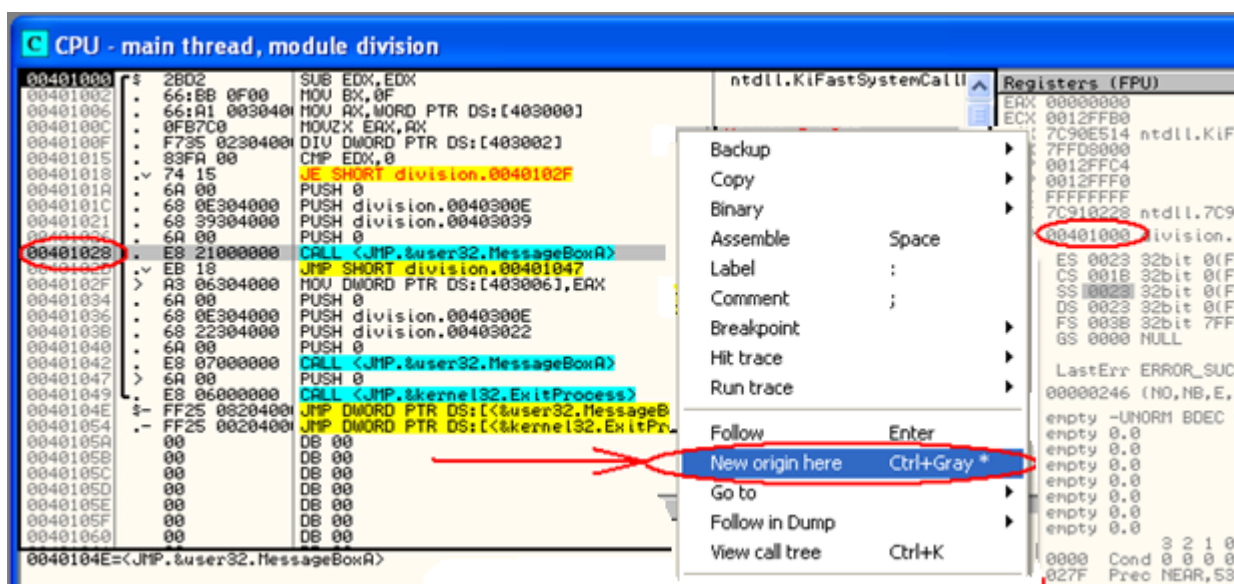
Содержимое всех остальных форматов скорректируется автоматически.

После нажатия кнопки «ОК» OllyDbg выделит изменённые значения красным цветом.

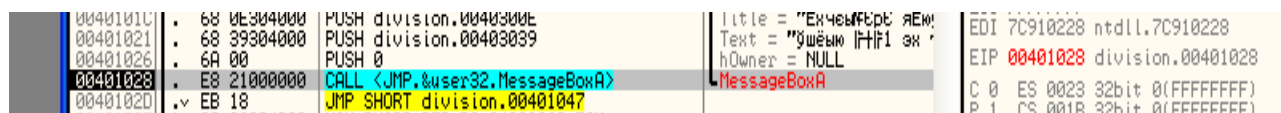


При желании можно вернуть старые значения. Для этого нужно кликнуть правой кнопкой мыши на поле выбранного регистра и в выпадающем меню выбрать опцию «Undo».

Таким образом, можно изменить содержимое всех регистров, кроме EIP. Этот регистр указывает на команду, которая должна выполниться следующей (на рис. ниже – EIP=00401000). Чтобы изменить содержимое EIP, а значит и порядок выполнения команд программы, сделайте следующее: выбираем новую команду в тексте, кликнув соответствующую строку в тексте программы или адрес, затем на ней кликаем правой кнопкой мыши и в выпадающем меню выбираем опцию “New origin here”.



Тогда EIP изменится на адрес выделенной строки, а программа продолжит выполнение именно с этого места. Как видим, теперь EIP = 00401028.



Содержимое сегмента данных (нижнее левое окно) или дампа памяти можно просматривать в различных форматах отображения:

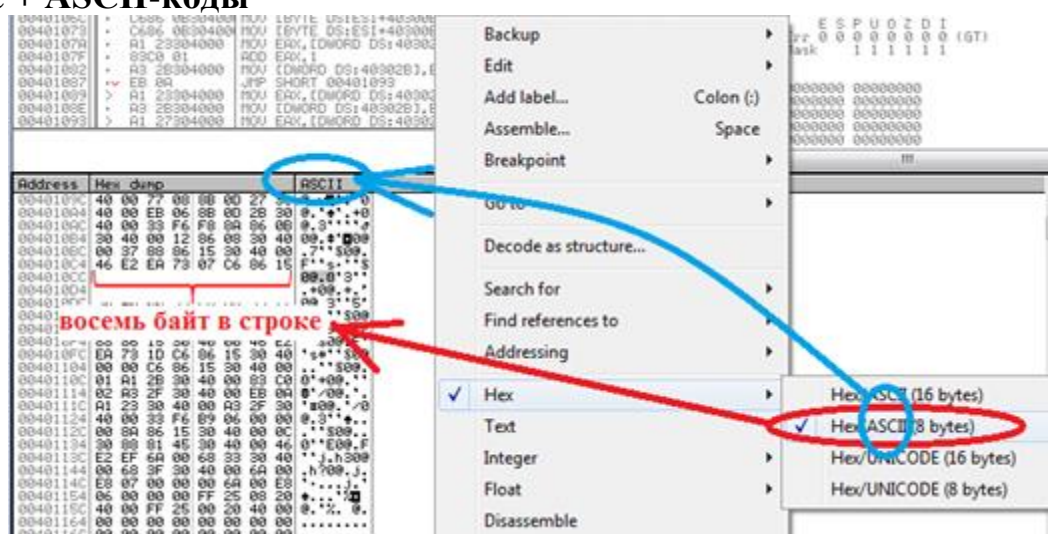
- текстовое представление (символы ASCII или UNICODE)
- побайтное представление в 16-шестнадцатеричных кодах

с. целых 8/16/32-разрядных чисел в знаковой/беззнаковой десятичной и шестнадцатеричной форме

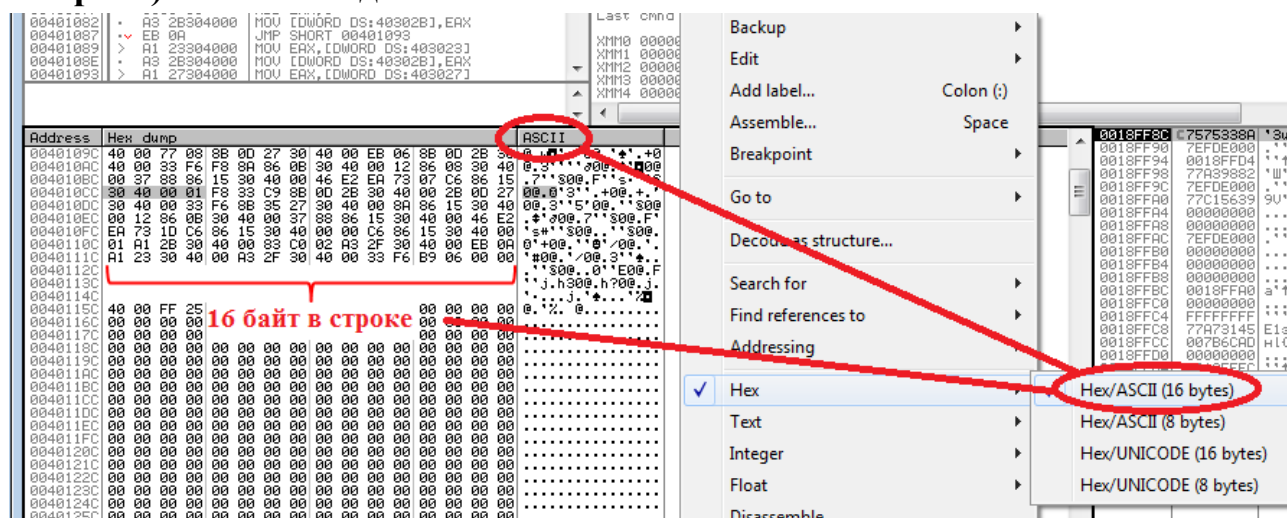
d. 32/64/80-разрядных дробных чисел в формате с плавающей запятой

Для изменения режима просмотра нужно кликнуть правой кнопкой мыши в любом месте поля дампа памяти и в выпадающем окне выбрать соответствующий режим интерпретации двоичных кодов ячеек памяти. Существует несколько режимов для просмотра HEX-кодов, целых и дробных чисел. Ниже приведены несколько примеров отображения данных в памяти.

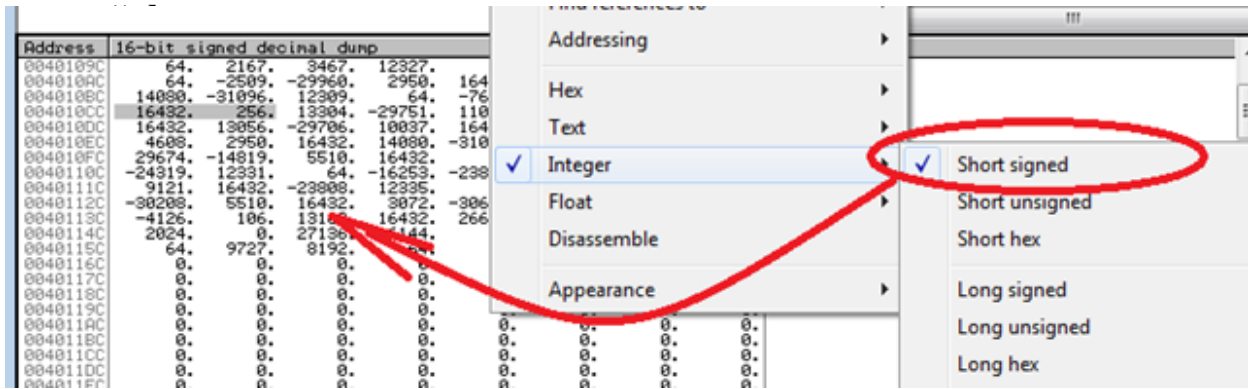
### Побайтное представление в 16-шестнадцатеричных кодах (HEX) по 8 байт в строке + ASCII-коды



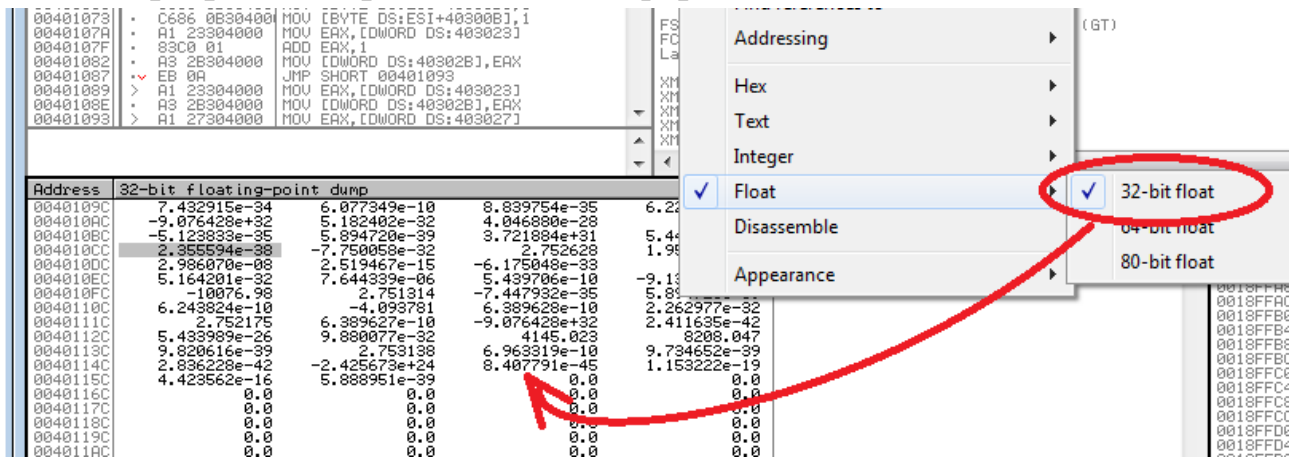
### Побайтное представление в 16-шестнадцатеричных кодах (HEX) по 16 байт в строке) + ASCII-коды



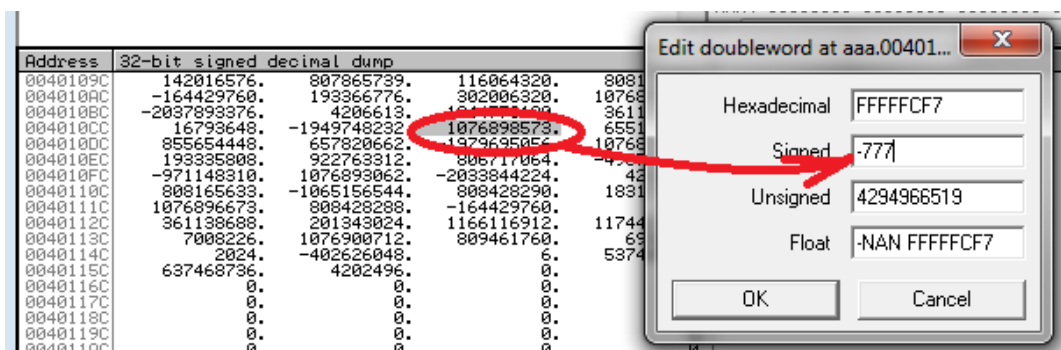
## Целых 8/16/32-разрядных чисел в знаковой/беззнаковой и шестнадцате-



## 32/64/80-разрядных дробных чисел в формате с плавающей запятой



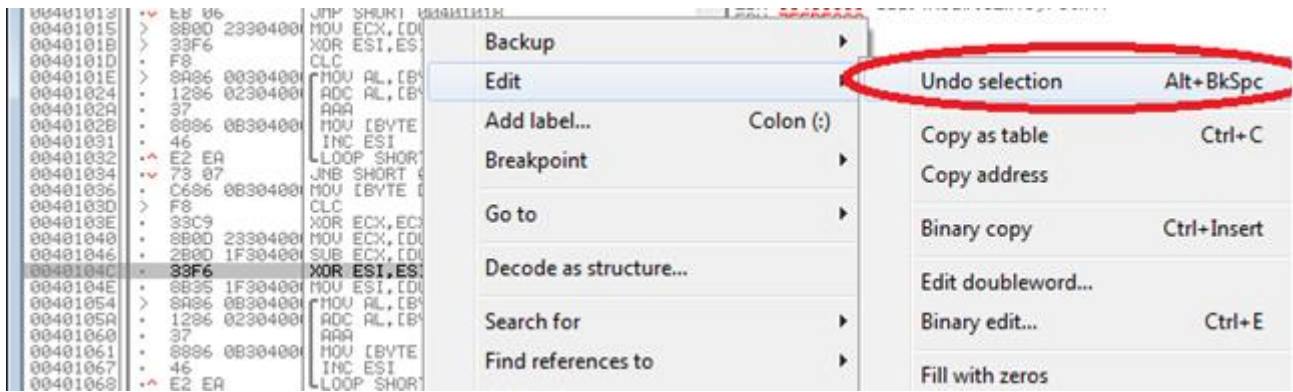
В ячейку памяти внести изменения очень просто. Надо просто выделить мышкой нужную ячейку и сразу начать набивать новое значение. В появившемся окне вы увидите вводимое значение в нескольких форматах представления сразу. После изменения ячейка окрашивается красным.



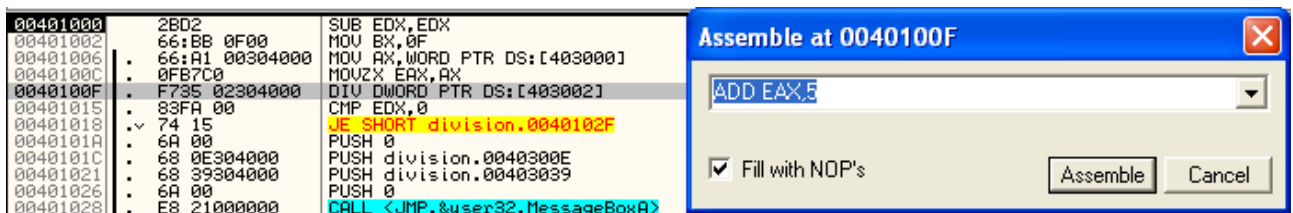
Для возврата старого значения кликнуть правой кнопкой мыши на поле изменённой (подсвеченной красным) ячейки, и в выпадающем меню выбрать опцию «EDIT» → «Undo selection» или воспользоваться сочетанием клавиш Alt+BkSpс.

Иванова Е. М. Учебное пособие по разделу «Вычислительные системы»

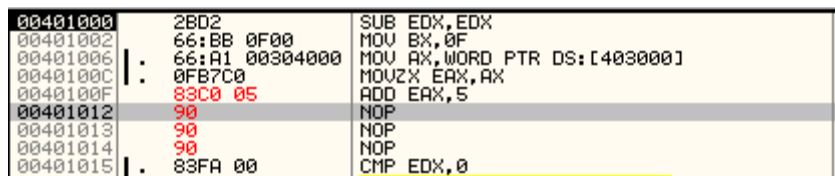




Также легко можно изменить любую команду в программе. Для этого надо выделить мышкой команду на замену, кликнуть правой кнопкой мыши на выделенном поле, в выпадающем меню выбрать опцию «Assemble» и в открывшемся окне ввести новую команду на Ассемблере. Затем нажать кнопку «Assemble».



Коды измененной области команд будет подсвечен красным.



Для возврата к исходному коду выделить эту строку (или строки) с красным адресом и кликнуть правой кнопкой мыши на выделенном поле. В выпадающем меню выбрать опцию «Undo selection».

*Надо помнить, что размер старой и новой команды должны быть одинаковыми, или новая должна быть меньше, тогда установить флажок «Fill with NOP's». **В противном случае изменения затронут и следующие байты памяти программ, т.е. следующие команды программы.***

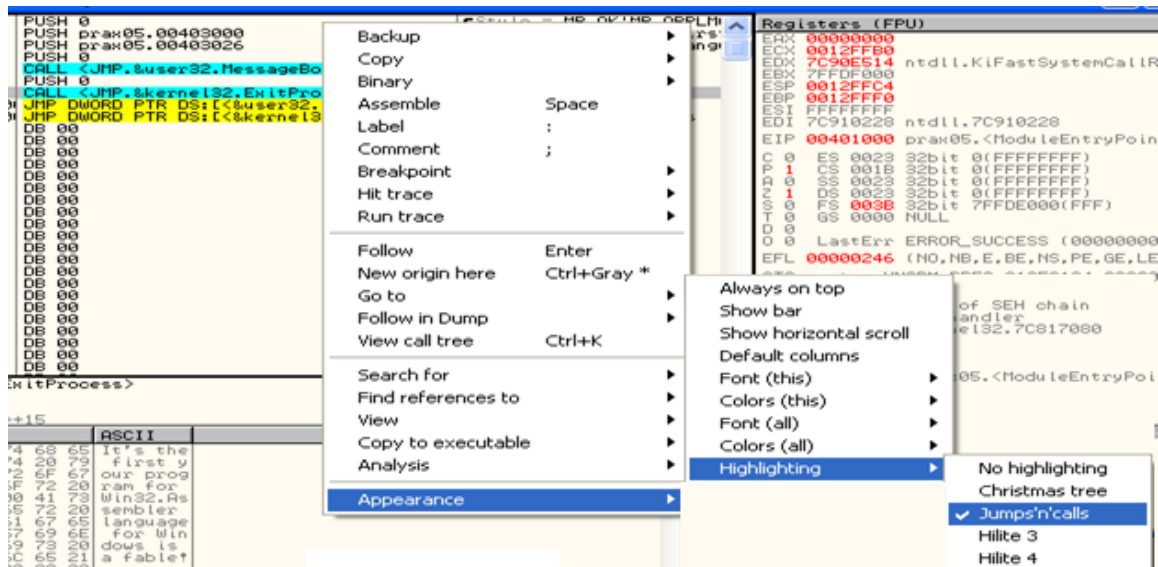
Другой опцией, которой вы можете воспользоваться для облегчения анализа программы, является подсветка **jmp** (переходов) и **call** (вызовов подпрограмм) – кликните в поле Ассембла правой кнопкой мыши и выберите

«Appearance -> Highlighting -> Jumps'n'Calls».

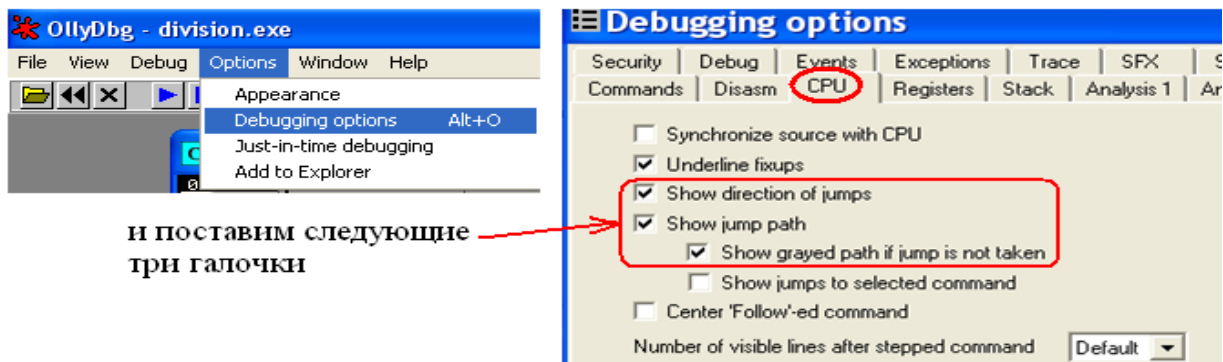
Получится следующее: команды CALL подсвечены лазурным цветом, а JMP – жёлтым, и текст программы стал более читаемым.

Для отмены подсветки надо кликнуть в поле Ассембла правой кнопкой мыши и выбрать

«Appearance -> Highlighting -> No highlighting».



Кроме того, можно сделать переходы в программе более наглядными. Если зайдём в опцию меню «Options/Debugging options», а там во вкладку CPU, где поставим следующие галочки



и поставим следующие три галочки

Теперь мы видим, что красной линией показано, откуда и куда совершается переход, в данном случае с адреса 00401000 на команду с адресом 00401031.

```

00401000 $v EB 2F JMP SHORT CRACKME.00401031
00401002 . E8 FF040000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007 . A3 CA204000 MOV DWORD PTR DS:[4020CA],EAX
0040100C . 6A 00 PUSH 0
0040100E . 68 F4204000 PUSH CRACKME.004020F4
00401013 . E8 A6040000 CALL <JMP.&USER32.FindWindowA>
00401018 . 0BC0 OR EAX,EAX
0040101A . 74 01 JE SHORT CRACKME.0040101D
0040101C . C3 RETN
0040101D > C705 64204000 MOV DWORD PTR DS:[402064],4003
00401027 > C705 68204000 MOV DWORD PTR DS:[402068],CRACKME.WndPr
00401031 > C705 6C204000 MOV DWORD PTR DS:[40206C],0
0040103B . C705 70204000 MOV DWORD PTR DS:[402070],0
00401045 . A1 CA204000 MOV EAX,DWORD PTR DS:[4020CA]
0040104A . A3 74204000 MOV DWORD PTR DS:[402074],EAX

```

В Ассемблере можно указывать числовые данные в системе счисления с основанием 16, 8, 2, 10. Однако в отладчике будут показаны их машинные коды в 16-чной системе счисления. Ниже показан пример задания одинакового по значению числа разными способами (обратите внимание на последний символ константы).

VAR DB 76d ; инициализация однобайтовой переменной VAR десятичной константой 76

VAR DB 4Ch ; --"--"-- шестнадцатеричной константой 4C

VAR DB 01001100b ; --"--"-- двоичной константой 01001100

VAR DB 114o ; --"--"-- восьмеричной константой 114

Кроме того, при указании шестнадцатеричной константы, начинающейся с буквы, требуется задавать дополнительный ноль вначале числа (255=0FFh). Это связано с необходимостью различать имена (которые могут содержать как буквы, так и цифры, но не должны начинаться с цифры) и шестнадцатеричной константы (которые также могут содержать как буквы, так и цифры, но должны начинаться с цифры).

Поскольку в приведённых записях все числа являются ASCII-кодом символа «L», то данная команда может быть записана ещё и так

VAR DB 'L' или VAR DB "L"

Попробуйте в вашей программе все способы задания числовых и символьных констант).

Из таблицы ниже видно, что код символа «L» равен шестнадцатеричному числу 4C<sub>16</sub> (=4×16<sup>1</sup>+12×16<sup>0</sup>=64+12=76<sub>10</sub>).

## ТАБЛИЦА ASCII КОДОВ

Основная таблица ASCII

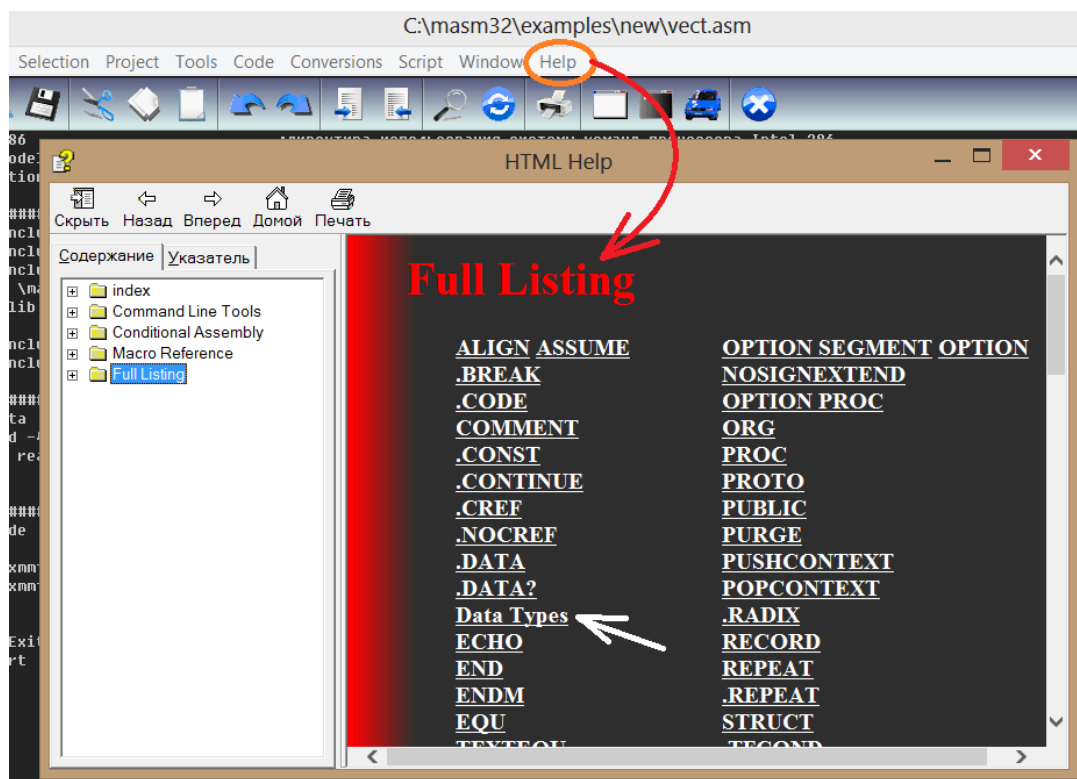
	00	10	20	30	40	50	60	70
0		▸		0	Q	P	,	p
1	☒	◀	!	1	A	Q	a	q
2	☒	⬆	"	2	B	R	b	r
3	♥	!!	#	3	C	S	c	s
4	♦	¶	\$	4	D	T	d	t
5	♣	§	%	5	E	U	e	u
6	♠	-	&	6	F	V	f	v
7	•	±	'	7	G	W	g	w
8	▣	↑	<	8	H	X	h	x
9	○	↓	>	9	I	Y	i	y
A	☒	→	*	:	J	Z	j	z
B	♂	←	+	;	K	[	k	{
C	♀	↳	,	<	L	\	l	
D	♂	↔	-	=	M	]	m	}
E	♀	⬆	.	>	N	^	n	~
F	♂	⬇	/	?	O	_	o	Δ

Расширенная таблица ASCII

	80	90	A0	B0	C0	D0	E0	F0
0	А	Р	а	▣	↳	⬆	Р	≡
1	Б	С	б	▣	↳	⬆	С	±
2	В	Т	в	▣	↳	⬆	Т	>
3	Г	У	г		↳	⬆	У	<
4	Д	Ф	д	↳	-	↳	Ф	†
5	Е	Х	е	↳	+	↳	Х	‡
6	Ж	Ц	ж	↳	↳	↳	Ц	‡
7	З	Ч	з	↳	↳	↳	Ч	≈
8	И	Ш	и	↳	↳	↳	Ш	°
9	Й	Щ	й	↳	↳	↳	Щ	·
A	К	Ь	к	↳	↳	↳	Ь	·
B	Л	Ы	л	↳	↳	↳	Ы	‡
C	М	Ь	м	↳	↳	↳	Ь	‡
D	Н	Э	н	↳	↳	↳	Э	²
E	О	Ю	о	↳	↳	↳	Ю	●
F	П	Я	п	↳	↳	↳	Я	

### 2.5. Форматы данных в Ассемблере Intel

Вы можете увидеть допустимые типы и форматы данных при просмотре встроенного справочника в MASM32.



Это следующие типы:

### Integer – целые

Формат	Обозначение/директива	Размер	Диапазон
BYTE	DB	1 Байт	$-128 \div 127$
WORD	DW	2 байта	$-32768 \div 65535$
DWORD	DD	4 Байта	$-2^{15} \div 2^{15} - 1$
FWORD	DF	6 Байт	$-2^{15} \div 2^{15} - 1$
QWORD	DQ	8 Байт	
TBYTE	DT	10 Байт	

### FP – дробные в формате ЧПЗ

Формат	Обозначение/директива	Размер	Представление
REAL4	DD	4 Байта	ЧПЗ одинарной точности
REAL8	DQ	8 Байт	ЧПЗ двойной точности
REAL10	DT	10 Байт	ЧПЗ расширенной точности

## 2.6. Пример программы

Ниже рассмотрен пример программы **prakt1.asm** для Win32, написанной под MASM. В программе две секции: данных (.data) и кода (.code). В секции данных размещены 5 переменных, которые в последствии будут размещены в оперативной памяти компьютера. Три из них символьные строки (header, mes\_Yes, mes\_No) используются для вывода в окне сообщения, и две целочисленные переменные (num1 и num2) для арифметических операций. Строковые данные определены побайтно (1 байт на каждый символ – согласно кодировке ASCII), о чём говорит директива **DB (Define Byte)**. Числовые переменные определены как двухбайтовые согласно директиве **DW (Define Word)** – слово у процессора Intel считается равным двум байтам. Можно указывать и другие размеры данных (см. выше).

Эта программа проверяет кратность двухбайтового числа num1 двухбайтовому числу num2 и выводит соответствующее сообщение. Для вывода сообщения, как и в предыдущем примере программы, воспользуемся API-функцией

### invoke MessageBox

```

.386
.model flat, stdcall
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
    num1        DW    220 ;переменная в памяти размером Word=16 бит=2 Байт
    num2        DW    20  ;переменная в памяти размером 2 Байт
    header      DB    "результат программы",0 ;строка в памяти, заданная побайтно (db)
    mes_Yes     DB    "число MEM1 кратно MEM2",0
    mes_No      DB    "число MEM1 не кратно MEM2",0
.code
start:          ;Метка начала программы
    SUB        DX,DX        ;Вычитание содержимого регистра DX из самого себя (обнуление).
                    ;Результат помещается на место первого операнда: DX=DX-EDX.
    MOV        AX,num1     ;Занесение содержимого из ячейки ОП с адресом MEM1 в регистр AX.
    DIV        num2       ;Деление 4 байт (DX:AX) на 2 байта из ОП (MEM2). AX – частное, DX - остаток.
    CMP        DX,0       ;Сравнение DX с нулём и установка флагов (равно, тогда EFL.ZF=1)
    JZ         ZERO      ;Условный переход по равенству (при ZF=1) на метку ZERO (кратно),
                    ;иначе (не равно, не кратно) переход к следующей команде программы.
    invoke     MessageBox, NULL, addr mes_No, addr header, MB_OK
    JMP       STOP        ;Безусловный переход на метку STOP.
ZERO: invoke  MessageBox, NULL, addr mes_Yes, addr header, MB_OK
STOP: invoke  ExitProcess, NULL ;Вызов процедуры, завершающей работу программы
end start        ;Конец программы

```

Для проверки на кратность разделим num1 на num2 командой DIV (беззнаковое целочисленное) и сравним остаток (регистр DX) с нулём. Команда DIV (согласно [2]) имеет синтаксис

### DIV <делитель>

и работает следующим образом.

#### DIV делитель

DIV (DIVide unsigned) – беззнаковое деление операндов – *делимое* и *делитель*.

**Действие:** *делимое* задается неявно, и его размер зависит от размера *делителя*, который явно указывается в команде. Местоположения *делимого*, *делителя*, *частного* и *остатка*, в зависимости от их размера, показаны в следующей таблице.

Размер операнда	Делимое	Делитель	Частное	Остаток	Максимальное частное
Слово/байт	AX	r/m8	AL	AH	255
Дв. слово/слово	DX:AX	r/m16	AX	DX	65,535
Учетв. слово/дв. слово	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

**Флаги:** OF=? SF=? ZF=? AF=? PF=? CF=?

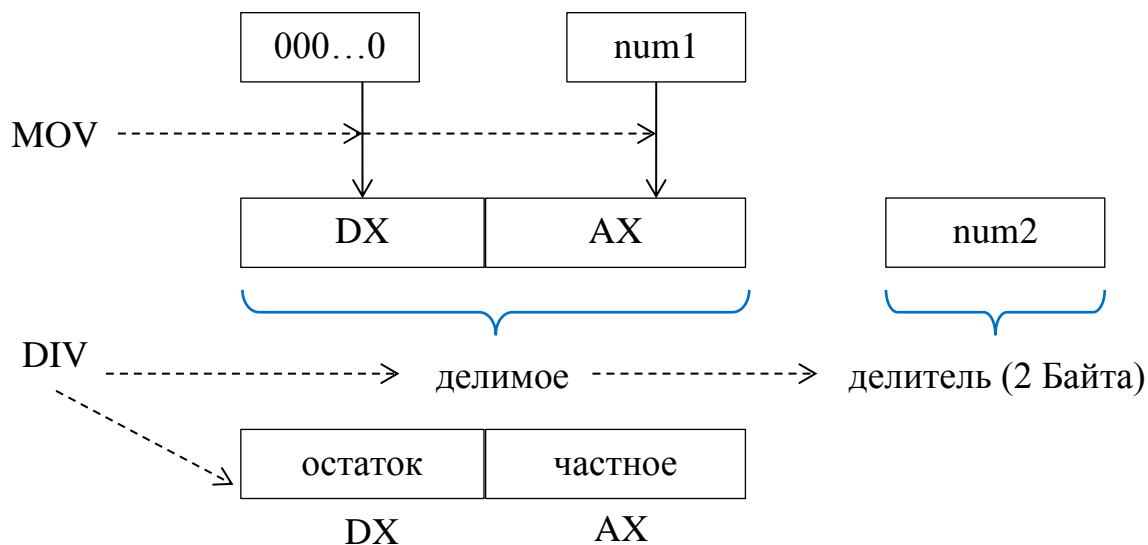
**Исключения:** PM: #DE; #GP(0): 2, 3; #PF(fault-code); #SS(0): 1; RM: #DE; #GP: 1; #SS: 1; VM: #AC(0); #DE; #GP(0): 1; #PF(fault-code); #SS(0): 1

Делитель указан в команде как единственный операнд, делимое должно быть заранее загружено в регистры процессора, какие именно определяем по таблице из описания. В нашем случае делитель имеет размер 2 байта, значит делимое должно быть расширено до 4 байт и помещено в пару регистров DX: AX. При расширении нужно оставить значение числа num1 неизменным, т.е. дополнить незначащими нулями слева. Для расширения числа со знаком (0 – положительное или 1 – отрицательное) надо в дополнительные старшие разряды дублировать знак числа.

В нашем примере числа беззнаковые, тогда старшая половина делимого (регистр DX) обнуляется. В регистр AX заносим num1.

Затем собственно деление и проверка остатка. Если остаток от деления равен нулю, значит число num1 разделилось нацело на num2, следовательно

выводим сообщение «кратно». Иначе выводим сообщение «не кратно». Текст этой программы приведён ниже.



Для тренировки можете поэкспериментировать с отрицательными/знаковыми числами ( $\pm 220 / \pm 20$ ) и соответствующими командами для обработки знаковых чисел (**IDIV**).

Следует отметить, что данный способ подходит для проверки кратности любых чисел, однако, если надо проверить кратность числу, являющемуся степенью двойки (2, 4, 8, 16...), то лучше воспользоваться другим методом. Можно вспомнить двоичное представление числа, в котором вес младшего правого разряда составляет «1». Тогда наличие единицы в этом разряде говорит о нечётности, а наличие нуля – о чётности.

5 = 0000010**1** – нечётное  
 6 = 0000011**0** – чётное  
 7 = 0000011**1** – нечётное  
 8 = 0000100**0** – чётное

Аналогично проверяется кратность 4, надо найти наличие двух младших нулей справа (их общий вес при отсутствии нулей = 1, 2 или 3).

4 = 000001**00** – кратно «4»  
 5 = 000001**01** – не кратно «4»  
 6 = 000001**10** – не кратно «4»  
 8 = 000010**00** – кратно «4»



Кратность 8 проверим наличием трёх младших нулей справа, кратность 16 – четырёх нулей и т.д.

## 2.7. Структура команды

Рассмотрим подробнее команды программы.

00401000	· 2BD2	SUB EDX,EDX
00401002	· 66:A1 00304000	MOV AX,WORD PTR DS:[403000]
00401008	· 0FB7C0	MOVZX EAX,AX
0040100B	· F735 02304000	DIV DWORD PTR DS:[403002]
00401011	· 83FA 00	CMP EDX,0
00401014	· 74 15	JE SHORT 0040102B
00401017	· 70 00	JMP 0

Первая исполняемая команда SUB EDX,EDX имеет длину два байта и шестнадцатеричный код 2BD2 (код первого байта 2B, код второго байта D2). Она располагается в памяти по адресу 00401000. У команды два операнда: первый – EDX, второй – EDX. Оба операнда расположены в регистре, адреса (№) регистров указываются в постбайте, расположенном сразу за байтом КОП (кода операции). Код команды SUB согласно [1,2] равен 2B и занимает 1 байт. Постбайт имеет следующую структуру: mod-reg1-reg2.

КОП	mod	постбайт	
2B	??	reg1 Код EDX	reg2 Код EDX

*В поле mod храниться некоторый код (??) режима адресации, какой конкретно не важно. Обратим внимание только на разряды, хранящие коды используемых командой регистров.*

Коды регистров (их порядковые номера) соответствуют расположению в окне OllyDbg

Registers (FPU)		№	(код)
EAX	770A3378 key	0	000
ECX	00000000	1	001
EDX	00000000	2	010
EBX	7EFDE000	3	011
ESP	0018FF8C	4	100
EBP	0018FF94	5	101
ESI	00000000	6	110
EDI	00000000	7	111

Значит содержимое постбайта будет следующим:

- в двоичном представлении – ??010010

- в шестнадцатеричном представлении – ?2

В нашем случае режим адресации имеет код 11 и мы получим содержимое постбайта 1101 0010 (D2).

Рассмотрим другую более сложную команду MOV AX,num1. Команда имеет длину шесть байт и код 66: A100304000. Она располагается в памяти по адресу 00401002.

У команды два операнда (и): первый – приёмник AX, второй – источник num1. Источник расположен в ОП по адресу 403000, приёмник – регистр AX. № регистра указываются в постбайте (последние 3 бита), адрес ячейки ОП в поле DISP (4 байта), расположенном после постбайта. Код команды MOV согласно [1,2] равен A1 и занимает 1 байт. Перед КОП расположен префикс изменения разрядности данных 16↔32 для текущей команды (по умолчанию настроено как 32 разряда, а в нашем случае размер операндов 16 разрядов – переменная num1 двухбайтовая). Постбайт имеет следующее содержимое: ?????000. Итого команда выглядит как

Префикс	КОП	mod	постбайт reg2	DISP
66	A1	?????	000 (код AX)	00 40 30 00

Однако в памяти компьютера многобайтовые числа (данные и адреса) располагаются согласно архитектуре Little Endian, начиная с младшего байта, т.е. поле DISP будет выглядеть 00 30 40 00. Что мы и видим на рисунке выше.

Рассмотрим ещё одну команду JZ ZERO. Команда имеет длину два байта и код 7415. Она располагается в памяти по адресу 00401014.

Это команда условного перехода (код операции 74) на метку ZERO, которая помечает первую команду API-функции invoke ..., расположенную по адресу 0040102B, что смещено относительно адреса текущей команды на величину:

$$0040102B - 00401014 = 17_{16} = 23_{10} \text{ байт.}$$

Вместо того, чтобы хранить длинный четырёхбайтовый абсолютный адрес перехода, в команде сохраняется этот короткое однобайтовое (SHORT)

смещение – относительный адрес перехода. Однако смещение указывается не относительно адреса текущей команды (который надо будет дополнительно вычислять), а относительно регистра IP-указателя следующей команды. Следующая за JZ ZERO команда имеет адрес, равный адрес (JZ ZERO) + длина (JZ ZERO) = 00401014+2=00401016. Поэтому итоговое смещение, составит

$$0040102B - 00401016 = 15_{16} = 21_{10} \text{ байт.}$$

Именно это шестнадцатеричное число и храниться в поле REL команды. Итого команда выглядит как

КОП REL

74 15

Что мы и видим на рисунке в начале раздела.

### 3. ЗАДАНИЕ

*Работа является индивидуальной и должна быть выполнена самостоятельно каждым студентом согласно варианту (=№ по списку группы).*

1. Научиться пользоваться пакетом MASM.
2. Создать программу **prakt1.asm** в редакторе Quick Editor пакета MASM.
3. Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторную компиляцию и запуск).
4. Внести в программу небольшие изменения (название окна с указанием вашей ФИО и № работы, выводимое сообщение, добавить вывод ещё одного окна ...) и запустить новую изменённую программу.
5. Научиться пользоваться пакетом OllyDbg.
6. Изменить программу **prakt1.asm** в редакторе Quick Editor пакета MASM согласно варианту задания. Придумать текстовые сообщения о результате программы и заменить в примере деление на одну или несколько других операций. Нужные команды на Ассемблере можно взять из [1, 2, 5].

7. Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторную компиляцию и запуск).
8. Убедиться, что в тексте программы нет синтаксических ошибок, и компиляция прошла успешно (в вашей папке появились два файла (prakt1.obj и prakt1.exe). если ошибки имеются, то нужно их исправить и повторить действия 3,4.
9. Запустить отладчик OllyDbg и открыть исполняемый файл **prakt1.exe**.
10. Отладить программу, используя раздел 2. и сайт [4], проследить изменения в регистрах.
11. Ответить на вопросы для самопроверки.

### 3.1. Варианты заданий

Вы должны уметь выполнить любое из заданий. Непосредственно сдаете написанную программу только по одному варианту и **разбираете форматы всех её команд**, но умеете ответить на вопрос: как выполнить любой из оставшихся вариантов заданий. Пусть в группе будет общий пул выполненных вариантов, надо понимать все. Ваш вариант задания только повод к разговору.

**ЕСЛИ ВЫ НЕ ЗНАЕТЕ, КАКИЕ КОМАНДЫ АССЕМБЛЕРА ВЫБРАТЬ – СПРОСИТЕ У ПРЕПОДАВАТЕЛЯ!!!**

№ вар	Задание
1.	Сравнить заданное число из ОП с некоторым порогом (константой) и вывести сообщение (равно, больше или меньше)
2.	Определить, является ли заданная в памяти переменная цифрой (заглавной или строчной латинской буквой) и вывести сообщение. Для любой десятичной цифры X её ASCII-код «3X» [см. табл. ASCII], значит надо сравнить старшие разряды с маской '0011' и вывести сообщение. Коды букв

	расположены в памяти последовательно с ХХ по УУ, следовательно достаточно понять, что код символа больше ХХ и меньше УУ.
3.	Сравнить два числа из ОП и вывести сообщение (равно, больше или меньше)
4.	Проверить, является ли число из ОП положительным или отрицательным, и вывести сообщение
5.	Определить минимальную/максимальную из трёх переменных и вывести сообщение
6.	Определить, превысила ли сумма трёх однобайтовых беззнаковых переменных размер в один байт (был ли получен перенос) и вывести сообщение.
7.	Проверить, допустима ли покупка нескольких товаров при заданном балансе счёта (не получается ли при вычитании отрицательное число) и вывести сообщение
8.	Определить есть ли среди нескольких чисел (переменные из ОП), совпадающие с заданным и вывести сообщение
9.	Проверить, являются ли два числа инверсными (not) и вывести сообщение
10.	Определить наличие единиц/нулей в двоичном представлении числа чётным и вывести сообщение, посчитать их количество
11.	Проверить выполнение условия $aa < bb * kk + cc - dd : ff$ , где $aa, bb, cc \dots$ - переменные и вывести сообщение
12.	Проверить выполнение условия, что операция логического ИЛИ над двумя переменными $aa \text{ } bb$ формирует все единичные биты результата и вывести сообщение

13.	Проверить выполнение условия, что операция логического И над двумя переменными <i>aa bb</i> формирует все нулевые биты результата и вывести сообщение
14.	Проверить значение <i>i</i> -ого слева бита в заданной двухбайтовой переменной (команда VT) и вывести сообщение
15.	Выполнить сдвиг в заданную сторону на заданное число разрядов. Проверить значение последнего выдвинутого бита и вывести сообщение (0 или 1).
16.	Проверить, что в заданных как ячейки памяти числах совпадают значения битов с №, заданным как ещё одна переменная.

#### 4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое машинный язык?
2. Что такое Ассемблер?
3. Что такое команда Ассемблера?
4. Что такое машинная команда?
5. Что такое директива?
6. Расскажите о порядке написания, компиляции и запуска программ в среде MASM.
7. Что такое qeditor.exe?
8. Что такое ml.exe?
9. Что такое link.exe?
10. Какими типами файлов оперирует программист при работе в пакете MASM?
11. Назовите назначение каждого из типов файлов (см.вопрос 10).
12. Зачем нужно изучать ассемблер?
13. Что такое мнемоника команды ассемблера?
14. Что такое отладчик OllyDbg?
15. Для чего нужен OllyDbg?

16. Расскажите об интерфейсе OllyDbg (назначение кнопок, опций, пиктограмм)?
17. Как проверить корректность работы программы для нескольких вариантов исходных данных без изменения исходного кода и без повторной компиляции?
18. Расскажите об изменениях в регистрах процессора по ходу выполнения вашей программы.
19. Что такое машинная команда?
20. Какова длина команды, как она определяется?
21. Как определить адрес текущей и следующей команды?
22. Что содержит регистр IP (EIP/RIP) процессора?
23. Из каких полей состоит команда (поясните на примере длинной команды из своей программы)?
24. Что такое мнемоника команды ассемблера?
25. Что такое операнды?
26. Какие операнды могут быть у команды (поясните на примере любых команд из своей программы)?
27. Где могут находиться операнды?
28. Как располагаются целые многобайтовые числа в памяти компьютера? Покажите пример такого числа и его размещения в ОП.
29. Расскажите о форматах представления целых чисел и приведите примеры из вашей программы размещения этих чисел в памяти с пояснением содержимого каждого байта числа.
30. Как работают команды передачи управления (условного перехода)? Приведите пример из своей программы.
31. Как можно проверить различные условия в программах на машинном языке?
32. Что такое флаги и флаговый регистр процессора?
33. Приведите пример использования значений флагов в своей программе.

## 5. СПОСОБ ОЦЕНИВАНИЯ

Максимальный балл 10 ставится студенту, правильно выполнившему задание, пояснившем правильно выполнение всех команд программы, их структуру и местоположение данных (операндов), ответившим верно на все контрольные вопросы.

Баллы снижаются за допущенные ошибки в программе, за неверные пояснения и ответы на вопросы.

## 6. СПИСОК ЛИТЕРАТУРЫ

1. 64-32-intel-software-developer-manual (с сайта intel.com)
2. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. – СПб.: Издательство «Питер», 2000.
3. Знакомство с MASM32 и Win32-программированием (<http://bitfry.narod.ru/07.htm>).
4. Рикардо Нарваха. Введение в крэкинг<sup>4</sup> с нуля, используя OllyDbg. Глава 1 (<http://pro.dtn.ru/cr.html>).
5. Юров В. Ассемблер. – СПб.: Издательство «Питер», 2002.
6. [ascii.org.ru](http://ascii.org.ru)

---

<sup>4</sup> — **Взлом программного обеспечения** (англ. *software cracking*) – действия, направленные на устранение защиты программного обеспечения (ПО), встроенной разработчиками для ограничения функциональных возможностей. Выполняется он именно на низком Ассемблерном уровне. Хотя сам сайт посвящён довольно специфической проблематике, но для решения задачи взлома требуется хорошо разбираться в особенностях машинного языка, и потому сайт содержит очень полезные комментарии и описания ПО, процессора, программ на Ассемблере и множество наглядных примеров.



## **ГЛАВА 2. ПРОГРАММИРОВАНИЕ В СРЕДЕ MARS.**

### **Практическая работа №2**

#### **1. ТЕОРИЯ**

##### **1.1.Машинный язык и Ассемблер**

В качестве основы для изучения выбран 32-разрядный RISC-процессор MIPS. Программы, написанные для этого типа процессоров, можно выполнять в моделирующем пакете MARS, который включает редактор и отладчик, демонстрирует наглядное представление команд и данных, их расположение в памяти, содержит справочную информацию как по самому пакету, так и по ассемблеру MIPS.

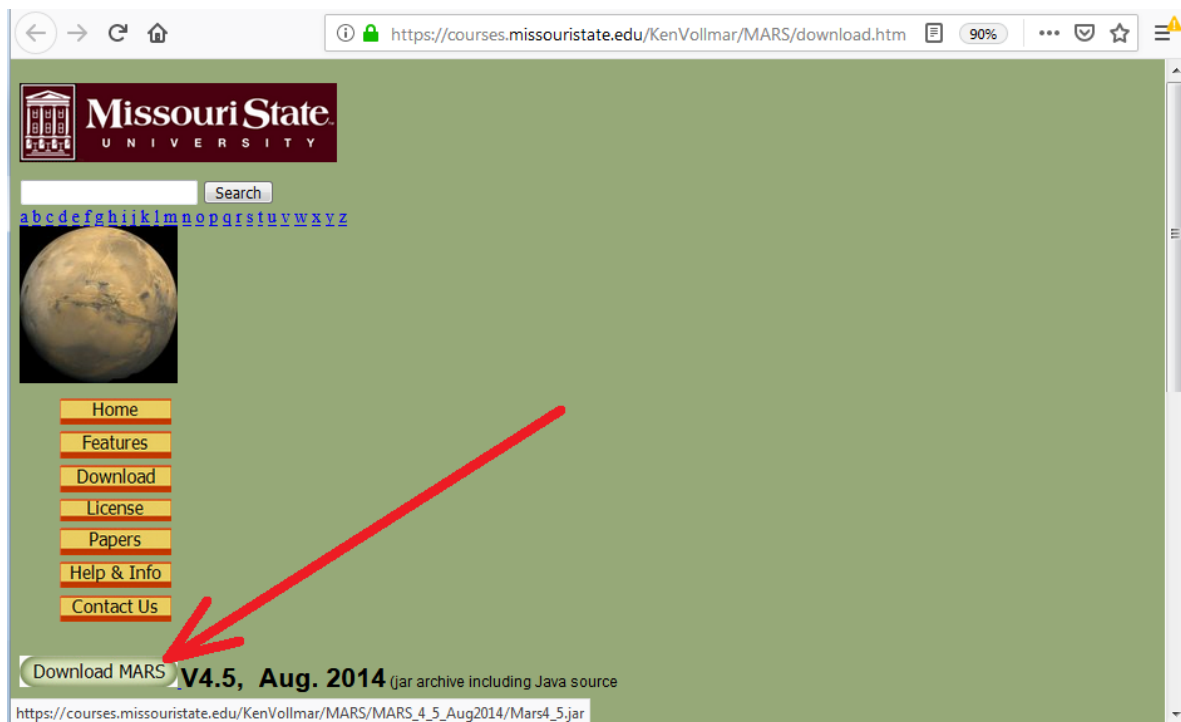
Машинный язык и Ассемблер для процессора с архитектурой MIPS гораздо проще: сокращено количество и разнообразие инструкций, которые имеют схожие мнемоники с Ассемблером Intel. Отличается структура программы, правила описания данных, имена/размеры/количество регистров (\$t0, \$f12), способ вывода результатов.

Для того чтобы хорошо разбираться во внутреннем строении компьютера и любой вычислительной системы и уметь её программировать нужно хорошо освоить уровень архитектуры набора команд, который представляет собой совокупность регистров, команд и других элементов, доступных программистам, пишущим на языках низкого уровня (системных программистов). Для первого знакомства с программированием на MIPS рассмотрим простейшую программу на Ассемблере. Для написания, транслирования и запуска будем использовать пакет MARS 4.5. Его файлы можно скачать, например, с сайта <https://courses.missouristate.edu/KenVollmar/MARS/download.htm>.

##### **1.2. Последовательность написания и отладки программ на Ассемблере**

Интерфейс пакета MARS стандартный и имеет привычные меню и горячие клавиши. Имеются две вкладки: EDIT и EXECUTE, которые позволяют соответственно создавать/редактировать файл, содержащий текст программы на Ассемблере и запускать его на исполнение/отлаживать с просмотром

содержимого регистров процессора, ячеек памяти, кодов команд программы и данных. Этот файл должен иметь расширение '.asm'. Когда исходный текст набран и сохранён, его можно ассемблировать (переводить на машинный язык) и запускать опцией меню «Run». Любые внесенные правки должны быть сначала сохранены и только потом запущены для исполнения.



Программа в машинных кодах готова непосредственно к выполнению в реальной аппаратной среде. Однако, в этом случае, обнаружение ошибок усложняется. Чтобы упростить задачу программиста, двоичную программу можно запустить не в реальной аппаратной среде, а в отладчике/симуляторе, который имеет ряд возможностей, в том числе и работать в режиме интерпретатора.



В последнем случае каждый отдельный момент времени выполняет только одну команду (интерпретирует) и выводит детальный отчет о своих действиях. В этом случае отлаживать программы становится значительно проще. В среде отладчика программы выполняются значительно медленнее, чем в реальных условиях. Пакет MARS имеет свой встроенный отладчик (вкладка “Execute”).

### 1.3. Программный пакет MARS. Установка и настройка


Установка не требуется. Скаченный файл является приложением java.

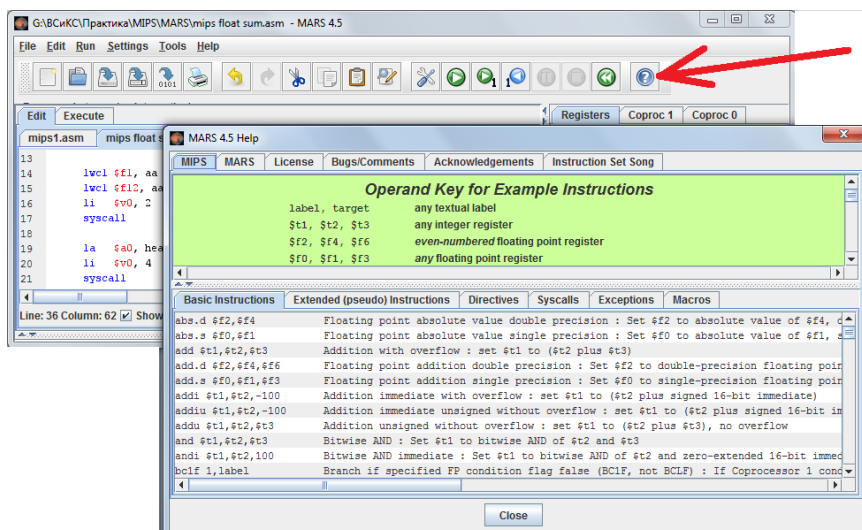
### 1.4. Написание программы на Ассемблере

Для написания программы нужно во вкладке EDIT набивать команды программы, либо вставить готовый текст программы, либо открыть заранее созданный файл.

Программа должна быть сохранена с расширением .asm (лучше в ту же папку, что и MARS).

### 1.5. Первый пример программы

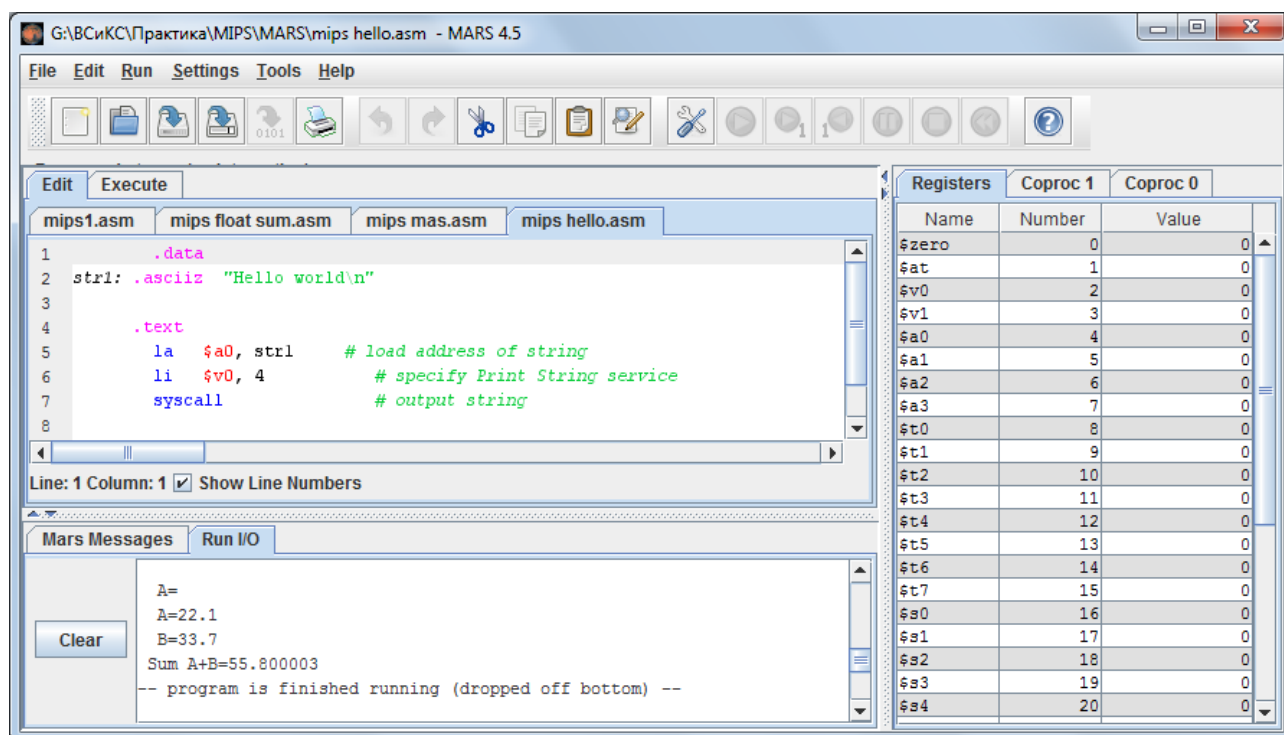
Ниже рассмотрен пример небольшой программы **mips\_hello.asm**. Для удобства написания и отладки в asm-программах кроме инструкций MIPS Ассемблера используются ещё так называемые псевдо-инструкции и системные вызовы. Все они описаны в меню помощи, встроенной в пакет и открывающейся по кнопке , либо в [2, 3, 4].



Программу **mips\_hello.asm** можно использовать как каркас любой вашей будущей MIPS-программы, далее можно вписывать туда только полезный код.

В ней всё, что располагается справа после знака # является комментарием и не транслируется в машинный код.

Более подробно описание работы пакета MARS можно найти там же во встроенном Help.



В окне пакета в верхней части расположены опции и кнопки меню. Их набор зависит от вкладки. Для вкладки «Edit» ниже в окне редактирования расположен текст программы, где разным цветом отмечены инструкции/системные сервисы, директивы, имена, комментарии, регистры. Программа состоит из двух секций: 1) `.data` – секция данных, 2) `.text` – секция кода/текста программы.

```
.data
str1: .asciiz "Hello world\n"

.text
la $a0, str1      # load address of string – псевдо-инструкция
li $v0, 4         # specify Print String service– псевдо-инструкция
syscall          # output string – системный сервис
```

В простой программе **mips\_hello.asm** в качестве данных используется текстовая строка для вывода сообщения, две псевдо-инструкции для

загрузки/перемещения данных и одна инструкция системного сервиса.

## 1.6. Задание № 1


1. Научиться пользоваться пакетом MARS.
2. Создать программу **mips\_hello.asm** во вкладке Edit пакета MARS.
3. Ассемблировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторное ассемблирование и запуск).
4. Внести в программу небольшие изменения (добавить вывод ещё одного сообщения) и запустить новую изменённую программу.

## 1.7. Порядок выполнения Задания № 1

1. Перейдите во вкладку редактирования (при запуске MARS открывается автоматически).

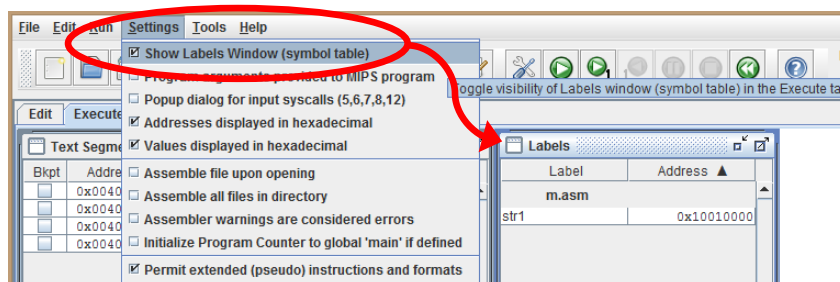
2. Наберите в ней текст программы или скопируйте из данной методички. Можно несколько строк набрать, а остальное скопировать, используя стандартную опцию меню “Edit/Paste”. Строки комментариев можно не набивать.

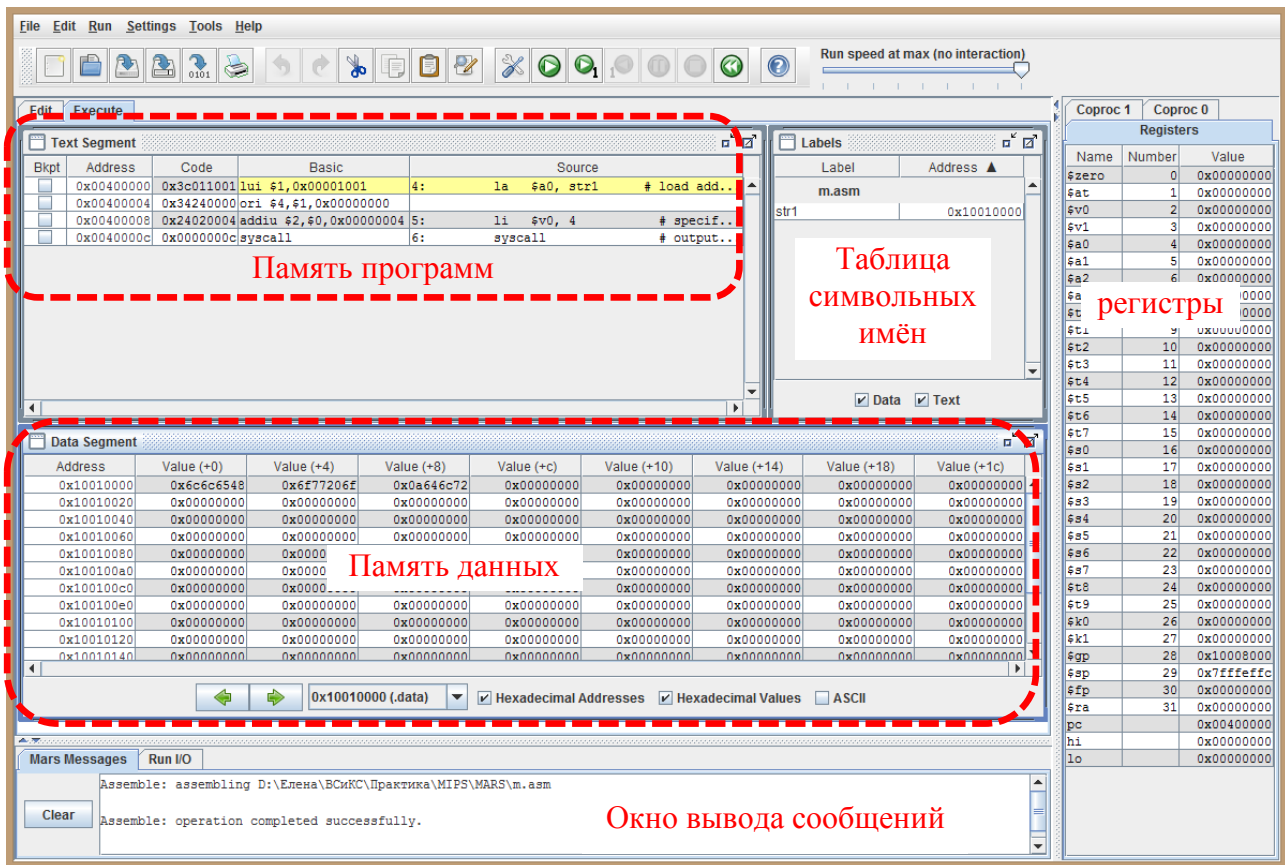
3. После всех изменений обязательно нужно сохранить файл, например опцией меню «File/Save as».

4. Для проверки ошибок и ассемблирования/компиляции нужно выполнить пункт меню Run > Assemble (горячая клавиша F3) или значок . При этом происходит переключение во вкладку “Execute”.

5. Убедитесь, что включена опция Меню «Settings / Show Labels Window (symbol table)», которая показывает, какой именно адрес соответствует имени

переменной/метки И упрощает поиск нужных переменных/команд В памяти программы или данных.





Здесь инструкции текстового сегмента программы показаны в виде таблицы с 5 столбцами:

- Bkpt (breakpoint) – в этом столбце можно поставить галочку, что будет означать точку останова при отладочном запуске,
- Address – в этом столбце указан адрес инструкции в ОП,
- Code – шестнадцатеричный код инструкции,
- Basic – базовые инструкции MIPS, в которые был транслирован исходный код программы (включая псевдо-инструкции),
- Source – исходный текст программы (с нумерацией строк).

В нижнем окне выводятся сообщения MARS и строковые переменные. Если текст программы не содержит ошибок, то в нём будут выведены сообщения:

```
Assemble: assembling G:\ВСиКК\Практика\MIPS\MARS\mips hello.asm
Assemble: operation completed successfully.
```


Если текст программы содержит синтаксические ошибки, то в окне будут выведены сообщения об ошибках с указанием № строки и позиции в строке, где именно предполагается ошибка и типа ошибки:

```
Assemble: assembling G:\ВСиК\Практика\MIPS\MARS\mips hello.asm
Error in G:\ВСиК\Практика\MIPS\MARS\mips hello.asm line 5 column 2: "la": Too few or incorrectly formatted operands. Expected: la $t1,($t2)
Assemble: operation completed with errors.
```


Исправьте ошибки и запустите компиляцию повторно.

5. Запустите программу-пример. Это можно сделать несколькими способами:


сразу целиком до точки останова

- через меню Run > Go,
- либо кнопкой ,
- либо клавишей F5;


по шагам

- через меню Run > Step,
- либо кнопкой  1 ,
- либо клавишей F7;

перезапустить программу с начала

- через меню Run > Reset,
- либо кнопкой ,
- либо клавишей F12.

Можно откатить выполнение программы на 1 шаг назад (множество раз)

- через меню Run > Backstep,
- либо кнопкой  1 ,
- либо клавишей F8.

6. Внесите изменения в текст программы. После внесения изменений в исходный файл перед его повторной компиляцией обязательно сохраните эти изменения в текущий файл опцией “Save” или в новый файл опцией “Save as”. И запустите изменённую программу.

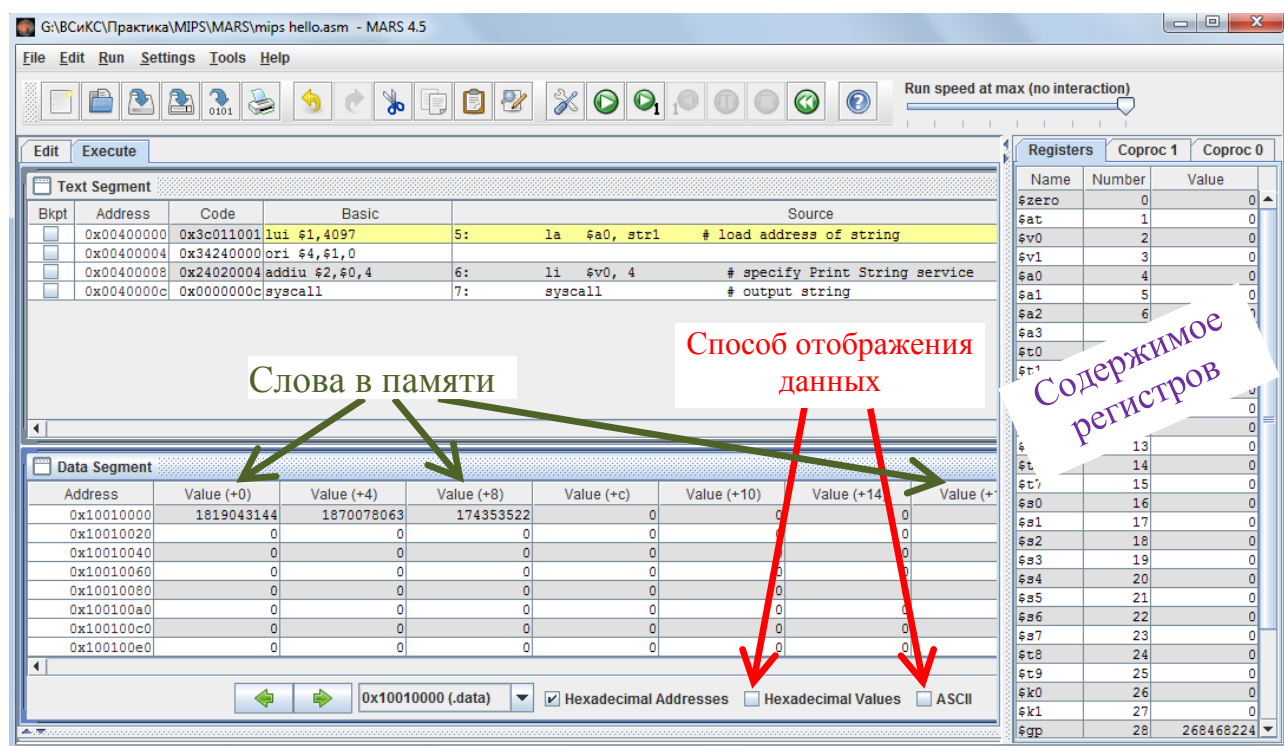
## 1.8. Опции отладки

MARS позволяет просмотреть содержимое ячеек памяти или регистров в 16-чном или 10-ичном формате, либо в виде ASCII символов (поставив галочку в соответствующем окошке – см. красная стрелка на рис. ниже).

Содержимое ячеек памяти отображается словами по 4 байта по 8 слов в строке с указанием адреса нулевого байта первого слова в начале каждой строки.

Размеры и границы окон могут быть изменены привычным для Windows способом.

Далее вы должны выполнять программу по шагам. За один шаг выполняется одна машинная команда. После каждого шага нужно отслеживать, как повлияла выполненная команда на состояние процессора и памяти (содержимое регистров, флагов, указателя адреса следующей команды PC, содержимое ячеек памяти по определенному адресу...).



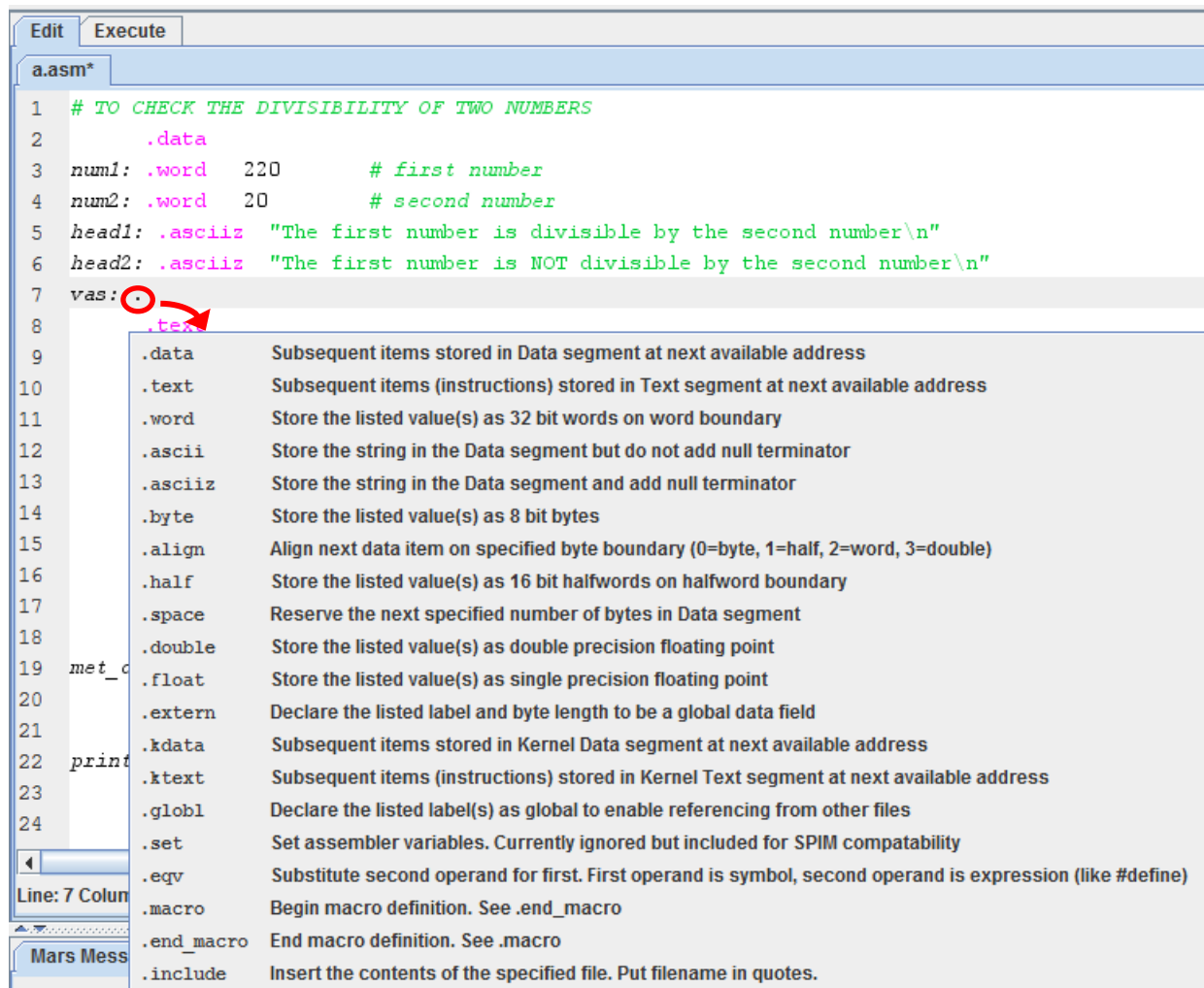
Для экономии места все ячейки ОП расположены как в таблице – построчно. Размер строки – 32 байт. В начале строки указан адрес первого в строке байта. Адреса остальных байтов при необходимости можно рассчитать



самостоятельно, используя шестнадцатеричную арифметику и указания в заголовках столбцов таблицы.

Содержимое любого регистра и любой ячейки памяти может быть изменено для проверки (в текст исходной программы на Ассемблере эти изменения не попадут). Для этого дважды кликнем левой кнопкой мыши на значении выбранного регистра/ячейки памяти и просто введём новое значение.

В MIPS Ассемблере можно указывать числовые данные в системе счисления с основанием 10 (`li $v0, 4`) и 16 (`li $v0, 0x4`). Допустимые форматы данных (`byte`, `word`, `half`...) можно увидеть в контекстной подсказке, когда вы ставите точку после имени и двоеточия.



В отличие от Ассемблера Intel в MIPS машинное слово имеет размер 32 бита, 16-битные данные называются полусловом (`.half`).

## 1.9. Пример программы

Ниже рассмотрен пример программы **prakt1\_MIPS.asm**, которая проверяет кратность числа `num1` числу `num2` и выводит соответствующее сообщение. Оба числа размером в слово – 32 бит.

В программе 2 секции: данных ( `.data`) и кода ( `.text`). в секции данных размещены 4 переменных: два числа `num1` и `num2`, кратность которых надо проверить, и две строки-сообщения.

Для проверки на кратность разделим `num1` и `num2` командой `DIVU` (беззнаковое целочисленное) и сравним остаток (регистр `HI`) с нулём. Сначала загружаем делимое `num1` в регистр `$t1`, а делитель – `num2` в `$t2`, оба имеют размер 4 байта. В результате деления содержимое регистров не изменится, но дополнительно установятся по результату два спец. регистра: `LO` – значением частного (`quotient`), `HI` – значением остатка (`remainder`). Проверив значение остатка, можно судить о кратности чисел и вывести соответствующее сообщение. Кроме того, две инструкции (`divu` и `mghi`) можно заменить на одну (`remu`).

Для тренировки можете поэкспериментировать со знаковыми числами (`-220 / -21`) и соответствующими командами для обработки знаковых чисел (`div` и `rem`), а также попробуйте обработать другие размеры чисел 2 или 1 байт.

## # TO CHECK THE DIVISIBILITY OF TWO NUMBERS

```
.data
MEM1: .word 220 # first number
MEM2: .word 20  # second number
head1: .asciiz "The first number is divisible by the second number\n"
head2: .asciiz "The first number is NOT divisible by the second number\n"

.text
la $t1, MEM1 # load address of the first number to register $t1
lw $t1, 0($t1) # load the first number to register $t1 (базовая адресация Адр=0+$t1)
lw $t2, MEM2 # load the second number to register $t2 - псевдоинструкция
divu $t1,$t2 # Divide unsigned $t1 by $t2 then set LO to quotient and HI to remainder
# (use mfhi to access HI, mflo to access LO)
mfhi $t3 # Move from LO register : Set $t3 to contents of HI=remainder
beq $t3,$zero,met_div # Branch if equal : Branch to statement at label's address if $t3 and NULLregister $zero are equal

la $a0, head2 # load address of print heading 'NOT DIV'
li $v0, 4 # specify Print String service = load immediate 4 to register $v0
j print_h # Jump unconditionally to statement at target address

met_div:
la $a0, head1 # load address of print heading 'DIVISIBLE'
li $v0, 4 # specify Print String service

print_h:
syscall # print heading

# две строки, выделенные жёлтым, можно заменить на одну:
#remu $t3,$t1,$t2 #REMAinder : Set $t3 to (remainder of $t1 divided by $t2, unsigned division)
```

## 1.10. Структура команды

Рассмотрим подробнее команды программы.

Address	Code	Basic	Source
0x00400000	0x3c011001	lui \$1,0x00001001	9: la \$t1, MEM1
0x00400004	0x34290000	ori \$9,\$1,0x00000000	
0x00400008	0x8d290000	lw \$9,0x00000000(\$9)	10: lw \$t1, 0(\$t1)

Исходный текст программы размещен в таблице в столбце «Source». Так можно увидеть, что некоторые исходные команды являются псевдоинструкциями и транслируются в другие реальные Ассемблерные инструкции MIPS. Первая команда программы

la \$t1, MEM1 # загрузить адрес MEM1 (константу 10010000) в регистр \$t1 таковой и является и преобразуется в две Ассемблерные исполняемые MIPS-инструкции

lui \$1,0x0001001 # загрузить старшие разряды адреса MEM1 в служебный адресный регистр \$at (его №=1) с добавленными нулями в младших байтах  
ori \$9,\$1,0x0000000 # «доклеить» младшую половину адреса = выполнить логическое ИЛИ с константой (где старшие 2 байта нулевые, а младшие разряды адреса = младшим разрядам адреса MEM1) и служебным адресным регистром \$at (его №=1), результат поместить в \$t1 – как запрограммировано в псевдоинструкции

Инструкция ori \$9,\$1,0x0000000 имеет длину четыре байта и шестнадцатеричный код 34290000 (см. столбец «Code»). Код первого байта 34, код второго байта 29, третьего и четвертого – 00). Она располагается в памяти по адресу 0040000 (см. столбец «Address»). Инструкция относится к типу I и имеет соответствующий формат (существует всего три типа форматов: R-тип, I-тип, J-тип).

### I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

32-битная команда состоит из четырёх полей: *op*, *rs*, *rt* и *imm*. Заметьте, что в архитектуре MIPS размеры полей команды не кратны байтам, а занимают произвольное число разрядов. Первые три поля (*op*, *rs* и *rt*) аналогичны таким же полям в командах типа R и содержат:

- *op*: код операции, *opcode*
- *rs*: номер регистра-источника (от 0 до 31)
- *rt*: номер регистра-назначения (от 0 до 31)
- *imm*: второй операнд-источник (16-битная знаковая константа)

В инструкциях I-типа выполняемая операция полностью определяется полем *opcode*. Операнды заданы в трёх полях: *rs*, *rt* и *imm*. В нашем примере:

формат	op	rs	rt	imm
разряды	0D	Регистр с № 1	Регистр с № 9	Константа 0
значение полей	001101000001010001	00000001	010001	0000000000000000
значение байтов	34		29	00 00

Поля *rs* и *imm* всегда используются как операнды-источники. Поле *rt* в некоторых командах (например, *addi* и *lw*) содержит номер регистра-назначения, в других (например, *sw*) – номер регистра-источника. Номера регистров можно увидеть в симуляторе MARS в таблице справа. Инструкции типа I содержат 16-битную константу *imm*, но константы участвуют в 32-битных операциях. 16-битные константы сначала будут расширены до 32 бит следующим образом: у неотрицательных констант верхние 16 бит будут заполнены нулями, а у отрицательных констант они будут заполнены единицами. Этот приём называется расширением знака. Расширение знака у числа, представленного в дополнительном коде, не меняет его значения. После расширения константы будет выполнена

Registers		
Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	4
\$v1	3	0
\$a0	4	268501000
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	0
\$t1	9	220
\$t2	10	20
\$t3	11	0
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$s0	16	0
\$s1	17	0
\$s2	18	0
\$s3	19	0
\$s4	20	0
\$s5	21	0
\$s6	22	0
\$s7	23	0
\$t8	24	0
\$t9	25	0
\$k0	26	0
\$k1	27	0
\$gp	28	268468224
\$sp	29	2147479548
\$fp	30	0
\$ra	31	0
pc		4194384
hi		0
lo		11

основная операция инструкции.

Рассмотрим другую команду `la $t1, MEM1`. Команда также является псевдо-инструкцией и при трансляции в Ассемблер превращается в 2 MIPS-инструкции:

`lui $1,4097` # загрузить константу=адрес переменной MEM1 в старшие 2 байта регистра №1, младшие два байта обнуляются  
`ori $9,$1,0` # логическое ИЛИ над регистром №1 и константой 0, результат в регистр №9 (\$t1)

Инструкция `lui $1,4097` имеет длину 4 байта и код 3c011001. Она располагается в памяти по адресу 0040004.

У команды фактически два операнда: первый – приёмник регистр №1, второй – источник константа 4097 (адрес переменной MEM1). Третьего регистра-источника нет, и он по умолчанию считается нулевым. Итого команда выглядит как

формат	op	rs	rt	imm
значение полей	Код операции	Регистр с № 0	Регистр с № 1	Константа 0
значение разрядов	0 0 1 1 1 1	0 0 0 0 0	0 0 0 0 1	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
значение байтов	3c		01	10 01

В памяти компьютера многобайтовые числа (данные и адреса) располагаются согласно архитектуре Little Endian, начиная с младшего байта, но поскольку симулятор MARS отображает ячейки 4-байтовыми словами, вы истинного порядка байтов в памяти не увидите.

Рассмотрим ещё одну команду `divu $t1,$t2`. Эта команда является Ассемблерной MIPS-инструкцией другого R-типа, имеет вид `divu $9,$10`.

## R-Type



В таких инструкциях значение поля `opcode=0` (значит инструкция R-типа) и операция определяется полем `funct`. Если поле `opcode` не нулевое, то это

инструкция другого типа. Назначение полей следующее:

- op: код операции, *opcode* (0 для всех инструкций R-типа)
- funct: функция (вместе с *opcode*, говорит компьютеру какую операцию необходимо выполнить)
- shamt: *shift amount*. Количество бит сдвига для инструкций сдвига (для остальных инструкций это поле 0)

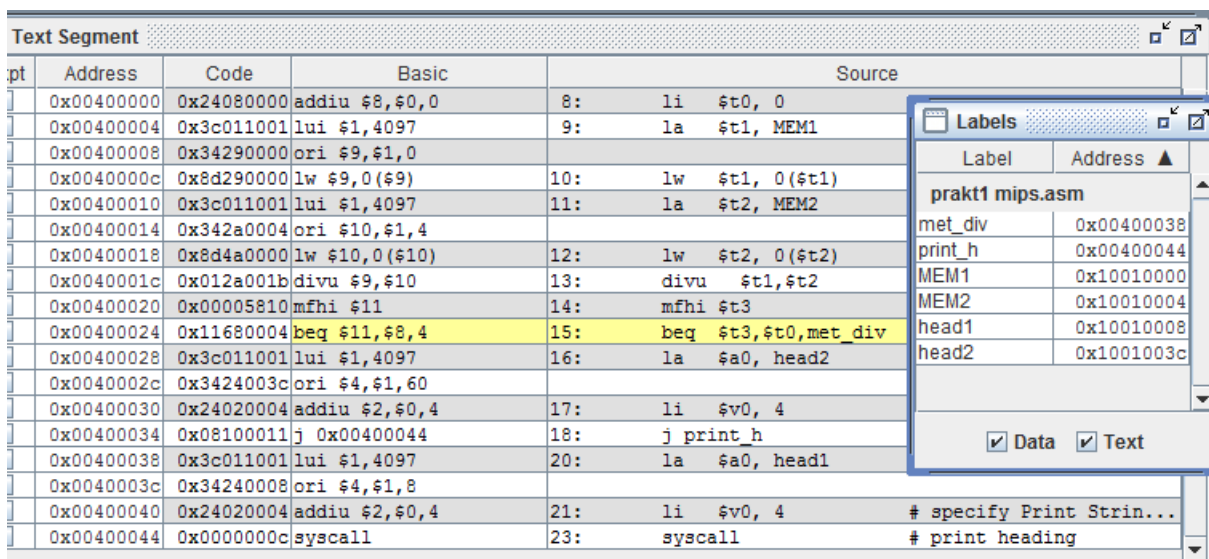
3 регистра-операнда:

- rs, rt: регистры-источники
- rd: регистр-назначение (регистр-приемник)
- 5-битовые поля rs, rt, rd содержат номера регистров (от 0 до 31)

Рассмотрим подробнее формат и код инструкции *divu \$9,\$10*.

формат	op					rs				rt				rd					shamt					funct									
значение разрядов	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1
значение байтов	01					2a				00					1b																		

Рассмотрим ещё одну команду *beq \$t3,\$t0,met\_div*. Эта команда является Ассемблерной MIPS-инструкцией I-типа, но в качестве одного из операндов – непосредственной константы X задан относительный адрес перехода на X слов/инструкций вперёд или назад (в зависимости от знака константы), что соответствует числу (X×4) Байт.

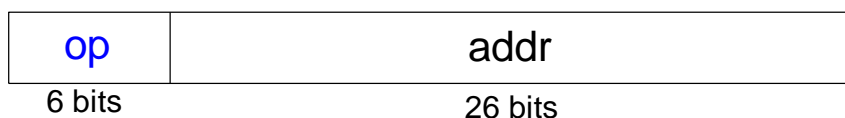


В нашем примере переход будет осуществляться на метку *met\_div*, что соответствует адресу 00400038. Адрес команды *beq* 0x00400024, однако во время её выполнения текущим адресом (который будет храниться в регистре РС-указателе команд становится адрес следующей команды т.е.  $0x00400024+4=0x00400028$ . Для перехода на метку нужно к текущему РС прибавить положительное число  $0x10=16$  Байт или 4 слова (константа  $X=4$ ).

формат	op	rs	rt	imm
значение разрядов	0 0 0 1 0 0	0 1 0 1 1	0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
значение байтов	11	68		00 04

Рассмотрим подробнее формат и код инструкции J-типа *j print\_h*. Формат такой инструкции простой

## J-Type



Здесь всего 2 поля:

- op – код операции,
- addr – адрес перехода (прямой)

При прямой (псевдопрямой) адресации адрес перехода задаётся внутри инструкции в поле «addr», размер которого 26 бит, что не достаточно для хранения полного прямого адреса перехода. К счастью, два младших бита адреса перехода ( $JTA1:0$ ) всегда должны быть равны нулю (их можно не хранить, а добавить в процессе выполнения), потому что все инструкции выровнены по словам. Следующие 26 бит адреса перехода (адрес метки  $[27-2]$ ) берутся из поля addr инструкции. Четыре старших бита адреса перехода ( $JTA31:28$ ) берутся из четырёх старших бит значения РС + 4 (адреса следующей команды). Такой способ адресации и называется псевдопрямым. В нашем примере инструкция *j print\_h* (расположенная по адресу 00400034) передает управление на адрес 00400044 (двоичное представление – 0000 0000 0100 0000 0000 0000 0100 0100



(выделены разряды с 27 по 2). В момент выполнения инструкции  $j$  РС содержит адрес 00400038 (старшие два бита – нулевые). Значит адрес перехода рассчитывается как:

$$\begin{aligned} \text{Адрес перехода} &= \text{РС}[31-30] \mid \text{адрес метки [27-2]} \mid 00 = \\ &= 00 \mid 0000\ 0100\ 0000\ 0000\ 0000\ 0100\ 01 \mid 00 \end{aligned}$$

Здесь синим показана константа, хранимая в поле «addr».

формат	op								addr																																
значение разрядов	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
значение байтов	08								10										00		11																				

В правильности расчётов можно убедиться, просмотрев коды в окнах симулятора.

Address	Code	Basic	Source
0x00400004	0x3c011001	lui \$1,4097	9: la \$t1, MEM1
0x00400008	0x34290000	ori \$9,\$1,0	
0x0040000c	0x8d290000	lw \$9,0(\$9)	10: lw \$t1, 0(\$t1)
0x00400010	0x3c011001	lui \$1,4097	11: la \$t2, MEM2
0x00400014	0x342a0004	ori \$10,\$1,4	
0x00400018	0x8d4a0000	lw \$10,0(\$10)	12: lw \$t2, 0(\$t2)
0x0040001c	0x012a001b	divu \$9,\$10	13: divu \$t1,\$t2
0x00400020	0x00005810	mfhi \$11	14: mfhi \$t3
0x00400024	0x11680004	beq \$11,\$8,4	15: beq \$t3,\$t0,met_div
0x00400028	0x3c011001	lui \$1,4097	16: la \$a0, head2
0x0040002c	0x3424003c	ori \$4,\$1,60	
0x00400030	0x24020004	addiu \$2,\$0,4	17: li \$v0, 4
0x00400034	0x08100011	j 0x00400044	18: j print_h
0x00400038	0x3c011001	lui \$1,4097	20: la \$a0, head1
0x0040003c	0x34240008	ori \$4,\$1,8	
0x00400040	0x24020004	addiu \$2,\$0,4	21: li \$v0, 4
0x00400044	0x0000000c	syscall	23: syscall

## 2. ЗАДАНИЕ

Работа является индивидуальной и должна быть выполнена самостоятельно каждым студентом согласно варианту (=№ по списку группы).

1. Научиться пользоваться пакетом MARS.
2. Создать программу для вывода сообщения.
3. Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторную компиляцию и запуск).

4. Внести в программу небольшие изменения (добавить вывод ещё одного сообщения) и запустить новую изменённую программу.
5. Изменить программу согласно варианту задания. Придумать текстовые сообщения о результате программы и заменить в примере деление на одну или несколько других операций. Нужные команды на Ассемблере можно взять из [2-4].
6. Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить ошибки и провести повторную компиляцию и запуск).
7. Отладить программу, проследить изменения в регистрах.
8. Ответить на вопросы для самопроверки.

### 3. ВАРИАНТЫ ЗАДАНИЙ

#### ЕСЛИ ПРОГРАММА ПРЕДПОЛОЖИТЕЛЬНО ЗАЙМЕТ ОБЪЁМ БОЛЕЕ 2 ЭКРАНОВ, ПРОКОНСУЛЬТИРУЙТЕСЬ С ПРЕПОДАВАТЕЛЕМ

Вы должны уметь выполнить любое из заданий. Непосредственно сдаете написанную программу только по одному варианту и разбираете форматы всех её команд, но умеете ответить на вопрос: как выполнить любой из оставшихся вариантов заданий. Пусть в группе будет общий пул выполненных вариантов, надо понимать все. Ваш вариант задания только повод к разговору.

#### ЕСЛИ ВЫ НЕ ЗНАЕТЕ, КАКИЕ КОМАНДЫ АССЕМБЛЕРА ВЫБРАТЬ – СПРОСИТЕ ПРЕПОДАВАТЕЛЯ !!!

№ вар	Задание
1.	Проверить значение $i$ -ого слева/справа бита в заданной одно-/двух-/четырёх-байтовой переменной из ОП и вывести сообщение (любой вариант формулировки на ваш выбор)
2.	Проверить выполнение условия, что операция логического ИЛИ над двумя переменными $aa\ bb$ формирует все единичные биты результата и вывести сообщение

3.	Проверить, является ли число из ОП положительным или отрицательным, и вывести сообщение
4.	Определить минимальную/максимальную из трёх переменных и вывести сообщение (любой вариант формулировки на ваш выбор)
5.	Проверить, являются ли два числа инверсными (not) и вывести сообщение
6.	Определить наличие единиц/нулей в двоичном представлении числа чётным и вывести сообщение, посчитать их количество
7.	Сравнить заданное число из ОП с некоторой константой и вывести сообщение (равно, больше или меньше)
8.	Сравнить два числа из ОП и вывести сообщение (равно, больше или меньше)
9.	Проверить выполнение условия $aa+bb*kk<cc-dd:ff$ , где $aa, bb, cc...$ - переменные и вывести сообщение
10.	Проверить выполнение условия, что операция логического И над двумя переменными $aa\ bb$ формирует все нулевые биты результата и вывести сообщение
11.	Выполнить сдвиг в заданную сторону на заданное число разрядов. Проверить значение последнего выдвинутого бита и вывести сообщение (0 или 1).
12.	Проверить, что в заданных как ячейки памяти числах совпадают значения битов с №, заданным как ещё одна переменная.
13.	Определить, превысила ли сумма двух однобайтовых беззнаковых переменных размер в один байт (был ли получен перенос) и вывести сообщение.
14.	Определить есть ли среди нескольких чисел (переменные из ОП), совпадающие с заданным и вывести сообщение

#### 4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое машинный язык?
2. Что такое Ассемблер?
3. Что такое инструкция Ассемблера?
4. Что такое машинная инструкция?
5. Что такое псевдо-инструкция?
6. Расскажите о порядке написания, компиляции и запуска программ в среде MARS.
7. Какими типами файлов оперирует программист при работе в пакете MARS?
8. Назовите назначение каждого из типов файлов (см.вопрос 10).
9. Зачем нужно изучать ассемблер?
10. Что такое мнемоника команды ассемблера?
11. Расскажите об интерфейсе MARS (назначение кнопок, опций, пиктограмм)?
12. Как проверить корректность работы программы для нескольких вариантов исходных данных без изменения исходного кода и без повторной компиляции?
13. Расскажите об изменениях в регистрах процессора по ходу выполнения вашей программы.
14. Какова длина инструкции, как она определяется?
15. Как определить адрес текущей и следующей команды?
16. Что содержит регистр РС процессора?
17. Из каких полей состоит команда (поясните на примере длинной команды из своей программы)?
18. Что такое операнды?
19. Какие операнды могут быть у команды (поясните на примере любых команд из своей программы)?
20. Где могут находиться операнды?
21. Как располагаются целые многобайтовые числа в памяти компьютера? Покажите пример такого числа и его размещения в ОП.

22. Расскажите о форматах представления целых чисел и приведите примеры из вашей программы размещения этих чисел в памяти с пояснением содержимого каждого байта числа. Покажите знак числа.
23. Как работают команды передачи управления (условного перехода)? Приведите пример из своей программы.
24. Как можно проверить различные условия в программах на машинном языке? Какие условия допустимы для проверки в MIPS?

## **5. СПОСОБ ОЦЕНИВАНИЯ**

Максимальный балл 10 ставится студенту, правильно выполнившему задание (для Intel и MIPS), пояснившем правильно выполнение всех команд программы, их структуру и местоположение данных (операндов), ответившим верно на все контрольные вопросы.

Баллы снижаются за допущенные ошибки в программе, за неверные пояснения и ответы на вопросы.

## **6. СПИСОК ЛИТЕРАТУРЫ**

1. <https://courses.missouristate.edu/KenVollmar/MARS/download.htm> или [https://github.com/MIPSfpga/schoolMIPS/tree/00\\_simple/scripts/bin](https://github.com/MIPSfpga/schoolMIPS/tree/00_simple/scripts/bin)
2. Хэррис, Д. М. Цифровая схемотехника и архитектура компьютера, 2017г (2019)
3. <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>.
4. Сайт «Учебный курс: Методы | Регистры MIPS»

# ГЛАВА 3. АДРЕСАЦИЯ ЭЛЕМЕНТОВ МАССИВА, ОРГАНИЗАЦИЯ ЦИКЛА. Практическая работа №3

## 1. РАБОТА С МАССИВАМИ В INTEL

### 1.1. Пример обращения к элементу одномерного массива

Для обращения к элементу одномерного массива в команде должна быть составлена формула для вычисления его адреса в памяти.

Адрес эл-та = адрес нач.массива + №эл-та×размер эл-та в байтах



Пример: рассчитать адрес 2<sup>го</sup> элемента.

$$\text{Адрес} = 0000 + 2 \times 2 = 0004$$

Имя массива однозначно определяет адрес младшего байта его нулевого элемента. Этот адрес можно загрузить в один из регистров общего назначения ЦП.

```
LEA ESI,A или MOV ESI,offset A
```

Если к содержимому этого регистра прибавить размер элемента в байтах

```
INC ESI или ADD ESI,2
```

то получим адрес следующего элемента, и так далее.

Для обращения по этому адресу надо записать следующую формулу:

```
MOV AX,[ESI] ;скопировать в регистр AX содержимое элемента  
;массива = ячейки памяти с адресом из регистра ESI
```

Или другой вариант

```
XOR ESI,ESI ;ESI=0  
MOV AX,A[ESI] ;AX=A[0]
```

## 1.2. Организация цикла for

Для перебора элементов массива можно организовать цикл. Количество повторений цикла (счётчик циклов) всегда задаётся в регистре ECX, а команда организации цикла LOOP уменьшает счётчик на 1 и проверяет получившееся значение на равенство 0. Если получен нулевой результат, то цикл заканчивается и управление передаётся следующей за LOOP командой. Иначе управление передаётся на начало цикла по адресу из адресной части команды LOOP.

```
.data
A    DD  -8, 3, -2, 4, 1 ;массив целых четырёхбайтовых чисел
k_neg DB  0             ;счётчик отрицательных элементов
k_pos DB  0             ;счётчик неотрицательных элементов
;-----
.code
start:  XOR ESI,ESI      ;ESI=0
        MOV ECX,5       ;ECX=5 счётчик циклов=число элементов
for1:   TEST A[ESI],0FFFFFFFH ;логическое И с числом 11...1
        JS metN         ;проверка знака результата (исходного числа)
                           ;и переход по знаку «минус» на метку metN
        INC k_pos       ;иначе увеличение счётчика неотрицат. на 1
        JMP NEXT1      ;переход к след. элементу массива
metN:   INC k_neg        ;увеличение счётчика отрицат. на 1
NEXT1:  ADD ESI,4        ;увеличение адреса на 4 – адрес начального
                           ;байта след. элемента в памяти
        LOOP for1      ;конец цикла
```

Допустимы и другие способы указания адреса ячейки памяти:

- $A[\text{рег}+\text{число}/\text{выражение}]$  ( $A[ESI+4]$ ),
- $A[\text{число}]$  ( $A[1]$ ),
- $A[\text{рег}+\text{рег}]$  ( $A[ESI+EBX]$ ),
- $[\text{offset } A+1*4]$ .

```

;стандартное начало программы
.data
    A            DW            0,1,2,4,6,7,3    ;массив двухбайтовых элементов
    ...

.code
    ...
    MOV    DL,1    ;флаг перестановок установить в «1» для первичного входа в цикл перебора
WHILE1:    ;начало цикла while
    MOV    ECX,7-1
    MOV    DL,0    ;сброс флага
    LEA   ESI,A    ;загрузка в регистр ESI адреса массива (нулевого элемента)
FOR1:     MOV    AX,[ESI]    ;загрузка в регистр AX очередного элемента массива с адресом ESI
    MOV    BX,[ESI+2]    ;загрузка в рег. BX след. по порядку элемента с адресом ESI+длина элемента (2)
    CMP   AX,BX    ;сравнение пары элементов
    JBE   NEXT1    ;AX≤BX тогда переход к следующей паре
    MOV   [ESI],BX ; } перестановка
    MOV   [ESI+2],AX ; }
    OR   DH,1    ;установка флага
NEXT1:    INC   ESI    ;вычисление адреса нового элемента для перебора ESI=ESI+длина эл-та
    INC   ESI
    LOOP FOR1    ;команда организации цикла попарных перестановок (со счётчиком ECX)
    OR   DH,0    ;проверка флага
    JNZ  WHILE1    ;на начало цикла перебора «пока флаг≠0» (без счётчика)

;вывод результатов
;стандартное завершение работы программы

```



### 1.3. Организация циклов while

Цикл while организуется с помощью команд условного перехода. Далее приведён фрагмент кода программы, которая сортирует одномерный массив по возрастанию методом пузырька. Этот метод подразумевает несколько переборов всех элементов. В каждом переборе происходит многократное попарное сравнение всех соседних элементов массива. Перед началом каждого перебора флаг перестановки сбрасывается (становится равным «0»). Если встречаются неупорядоченные пары, то элементы пары меняются местами и устанавливается флаг необходимости провести ещё один перебор. Процедура перебора повторяется до тех пор, пока флаг перестановки равен «1». Если после очередного перебора флаг остаётся нулевым, то сортировка заканчивается.

### 1.4. Использование двумерных массивов (матриц)

Все элементы двумерных массивов располагаются в памяти последовательно друг за другом, после последнего элемента первой строки следует первый элемент второй строки.

адрес байта ОП	Содержимое ячеек
40000	Элемент $A[0,0] = -1$
40001	
40002	
40003	
40004	Элемент $A[0,1] = -2$
40005	
40006	
40007	
40008	Элемент $A[0,2] = -3$
40009	
4000A	
4000B	
4000C	Элемент $A[0,3] = -4$
4000D	
4000E	
4000F	
40010	Элемент $A[1,0] = 1$
40011	
40012	
40013	
40014	Элемент $A[1,1] = 2$
40015	
40016	
40017	
40018	...

Пусть, например,

задана матрица  $3 \times 4$ :

-1,	-2,	-3,	-4,
1,	2,	0,	4,
2000,	4096,	65535,	0

## 1.5. Пример обращения к элементу двумерного массива

Для обращения к элементу двумерного массива требуется в команде закодировать формулу для вычисления его адреса в памяти.

$$\text{Адрес эл-та} = \left( \begin{array}{c} \text{адрес} \\ \text{начала} \\ \text{массива} \end{array} \right) + \text{№ строки} \times \left( \begin{array}{c} \text{размер} \\ \text{строки в} \\ \text{байтах} \end{array} \right) + \text{№ столбца} \times \left( \begin{array}{c} \text{размер} \\ \text{элемента} \\ \text{в байтах} \end{array} \right)$$

Пример: рассчитать адрес элемента A[1,1].

$$\text{Адр. } A[1,1] = 40000_{16} + 1 \times 10_{16} + 1 \times 4 = 40014_{16}$$

Адрес начала массива	№_строки	Размер строки в байтах	№_эл-та	Размер эл-та в байтах
----------------------	----------	------------------------	---------	-----------------------

Когда вы в программе будете обращаться к каждому элементу поочерёдно, то можно в качестве ссылки на строку и столбец матрицы использовать регистры, например, EBX и ESI. Тогда в качестве адреса 4-байтового элемента можно записать следующее выражение:

$$A[EBX+ESI*4]$$

где

A – абсолютный адрес начала массива A в памяти,

EBX – относительный адрес начала строки внутри массива (относительно начала массива),

ESI – относительный адрес столбца в строке (относительно начала строки),

4 – масштабный множитель, указывающий на размер элемента массива.

Для команды с таким режимом адресации используется формат:

КОП	постбайт	SIB	Disp
-----	----------	-----	------

На рисунке ниже показаны значения этих полей для команды, обращающейся к элементу матрицы.

Если в цикле нужно перебрать все элементы матрицы, то правильнее организовать два вложенных цикла: по строкам и по столбцам. Однако, следует помнить, что команда организации цикла LOOP работает только с регистром

ECX, а значит для каждого цикла надо использовать этот регистр своим значением, запоминая временно его текущее значение в переменной в памяти/в регистре/в стеке. (Представьте себе, что в Паскале/СИ аналогично можно было бы записать цикл *for* только с переменной *i* и для каждого нового вложенного цикла пришлось бы запоминать старое значение *i* и инициализировать новое *i*).

КОП  
постбайт  
Disp=адрес  
начала А  
константа

SIB  
B3 = 10110011  
Scale=10 (код масштабного множителя 4)  
Index=110 (код регистра ESI)  
Base=011 (код регистра EBX)

Registers (MMX)		№
EAX	76F73378	000
ECX	00000000	001
EDX	00401000	010
EBX	7EFDE000	011
ESP	0018FF9C	100
EBP	0018FF94	101
ESI	00000000	110
EDI	00000000	111

Также можно один из циклов организовать с помощью команды LOOP, а прочие с использованием команд условного перехода Jccs.

Пример программы, перебирающей поочередно все элементы двумерного массива приведён ниже.

### 1.7. Пример программы подсчёта количества нулевых элементов в двумерном массиве

```
.data
A          DD  -1, -2, -3, -4, 1, 2, 0, 4, 2000, 4096, 65535, 0
k_str      dD  3
k_stolb    dD  4
KOL_ZERO   Dd   0
Zagolovok  db  "programm result",0
sResult    byte 50 dup (?)
sfc        db  "в массиве А нулевых элементов - %.1li шт.",0
;-----
```

```

.code
start:      XOR EBX,EBX
           MOV ECX,k_str      ;счётчик циклов по строкам
for_STR:   MOV EDX,k_stolb   ;счётчик циклов по столбцам
           XOR ESI,ESI
           XCHG ECX,EDX      ;перенастройка ECX на счётчик столбцов
for_STOLB: TEST A[EBX+ESI*4],0FFFFFFFH
           JNZ NEXT_STOLB
           INC KOL_ZERO
NEXT_STOLB: INC ESI
           LOOP for_STOLB
           ADD EBX,16
           XCHG ECX,EDX      ;перенастройка ECX на счётчик строк
           LOOP for_STR
           invoke sprintf, ADDR sResult, ADDR sfc, KOL_ZERO
           invoke MessageBox, NULL, ADDR sResult, addr Zagolovok, MB_OK
           invoke ExitProcess, NULL
end start      ;Конец программы

```

### 1.8. Примеры различных режимов адресации операндов и команд

**XOR EBX,EBX** ;оба операнда в регистрах – режим адресации: явная, прямая

```

  330B | XOR EBX,EBX
  /  |  \
КОП  постбайт   код первого регистра, код второго регистра
11011011

```

**MOV ECX,k\_str** ;1<sup>ый</sup> операнд – регистр (адресация явная, прямая – код в постбайте)  
;2<sup>ой</sup> оп-д – ячейка ОП (адресация явная, прямая – адрес в поле Disp)

```

  8B0D 30304000 | MOV ECX, DWORD PTR DS:[403030]
  /  |  \
КОП  постбайт  Disp – 4б.
      00001101  в формате «little endian»
      код регистра ECX

```

**JNZ NEXT\_STOLB** ;операнд – относительный адрес перехода показывает  
;смещение относительно текущего EIP

```

0040101D | 75 06 | JNE SHORT 00401025
0040101F | FF05 38304000 | INC DWORD PTR DS:[403038]
00401025 | 46 | INC ESI

```

75 06 Когда выполняется команда JNZ с адресом 0040101D  
КОП относительный адрес EIP указывает на следующую команду и равен 0040101F. Чтобы выполнить переход на команду с адресом 00401025 нужно прибавить к текущему значению EIP число +6. Это и есть величина смещения, хранящаяся в соответствующем поле команды.

### ADD EBX,16

```

      83C3 10 | ADD EBX, 10
-----|-----
КОП Постбайт 11000011 Const – 16.
    – код регистра EBX в шестнадцатеричном формате

```

**LOOP for\_STOLB** ;операнд – относительный адрес перехода показывает  
;смещение относительно текущего EIP

```

00401012 | F784B3 00304000 FFFFFFFF | TEST DWORD PTR DS:[ESI*4+EBX+403000],FFFFFFF
0040101D | 75 06 | JNE SHORT 00401025
0040101F | FF05 38304000 | INC DWORD PTR DS:[403038]
00401025 | 46 | INC ESI
00401026 | E2 EA | LOOP SHORT 00401012

```

КОП относительный адрес

Когда выполняется команда LOOP с адресом 00401026 EIP указывает на следующую команду и равен 00401028. Чтобы выполнить переход на команду с адресом 00401012 нужно прибавить к текущему значению EIP число  $-16_{16}$ . Отрицательное число храниться в дополнительном коде:

$$[-16_{16}]_{\text{доп}} = [-0010110_2]_{\text{доп}} = 11101010_2 = EA_{16}$$

Это и есть величина смещения, хранящаяся в соответствующем поле команды.

## 1.9. API-функция sprintf

Для вывода числа в окно результата существует API-функция sprintf и другие. Она заносит число (или несколько чисел) в символьную строку. А затем, с помощью уже известной нам функции MessageBox мы можем вывести эту строку в окно на экране.

*invoke wsprintf, ADDR sResult, ADDR sfc, a, b, sum\_ab*

Здесь

- первый параметр (*ADDR sResult*) – адрес будущей строки, содержащей некоторый текст (по желанию текст может отсутствовать) и наше число/или числа
- второй параметр (*ADDR sfc*) – адрес строки формата,
- третий параметр (*a*) – число1,
- четвёртый параметр (*b*) – число2,
- пятый параметр (*sum\_ab*) – число3,
- ...

Рассмотрим теперь, как может выглядеть строка формата вывода трёх чисел *a*, *b* и их суммы *sum\_ab*.

*sfc db "Заданы a=%.8lX, b=%.8lX, ",10,"Их сумма равна%.8lX ", 0*

Здесь использованы следующие параметры:

“...” – всё, что стоит в кавычках, это текст, который попадёт на экран без изменений, за исключением некоторых специальных (служебных) символов (см. табл. ASCII). К таким символам, например, относятся

% – символ начала формата. До него может идти любой текст, который будет скопирован в строку результата неизменным.

.8 – символ точки и число за ним определяют, сколько разрядов мы хотим получить на выходе (пустые заполняются нулями).

lX – тип результирующего числа (long unsigned heXadecimal integer) 32-битное положительное шестнадцатеричное целое заглавными буквами.

*Код 10* – код служебного символа смены строки (чтобы сообщение выводилось в две строки)

0 – нулевой байт окончания строки.

Некоторые другие типы результирующего числа приведены в таблице ниже.

Описание типа	Предназначение
<i>C, c, hc, hC, lc, lC</i>	Единичный символ.
<i>d, i</i>	Знаковое десятичное целое число (signed integer).
<i>hd</i>	Знаковое короткое целое число (signed short integer).
<i>s, S, hs, hS, ls, lS</i>	Строка символов.
<i>hu</i>	Беззнаковое короткое целое число (unsigned short integer).
<i>ld, li</i>	Знаковое длинное целое число (signed long integer).
<i>lu</i>	Беззнаковое длинное целое число (unsigned long integer).
<i>lx, lX</i>	Длинное шестнадцатеричное целое число без знака в нижнем регистре (00fa) или верхнем регистре (00FA) (unsigned long integer).
<i>p</i>	<b>Windows 2000/XP:</b> Указатель. Адрес печатается, используя шестнадцатеричную систему.
<i>u</i>	Целое число без знака (unsigned integer).
<i>x, X</i>	Шестнадцатеричное целое число без знака в нижнем или верхнем регистре (unsigned integer).

Далее приведён пример программы, которая вычисляет сумму двух знаковых длинных целых чисел и выводит в окно результат вычислений.

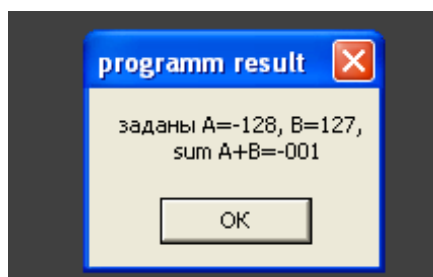
```
.data
A          DD    -128
B          DD     127
sumAB     DD     0
Zagolovok db    "programm result",0
sResult   byte 50 dup (?) ; выделение 50 байт под будущую строку с
; текстом и числами с неопределённым нач.значением каждого байта
sfc       db    "заданы A=%3li, B=%3li,",10,"      sum A+B=%3li",0
; Строка формата для API-функции wsprintf
```

```

;-----
.code
start:
    MOV EAX,A
    ADD EAX,B
    MOV sumAB,EAX
    invoke wsprintf, ADDR sResult, ADDR sfc, A, B, sumAB
    invoke MessageBox, NULL, ADDR sResult, addr Zagolovok, MB_OK
STOP:  invoke ExitProcess, NULL
end start      ;Конец программы

```

Эта программа даёт следующий результат:



Для вывода результатов работы вашей программы вы можете воспользоваться другими библиотечными функциями (в том числе другими API-функциями) при условии, что вы сможете объяснить порядок их применения. За самостоятельно найденное решения по организации ввода-вывода оценка повышается.

## 2 РАБОТА С МАССИВАМИ В MIPS

Рассмотрим последовательность написания и отладки программы на Ассемблере MIPS, оперирующей элементами массива и выполняющей циклы.

### 2.1.Обращение к элементу одномерного массива

Различные процессоры MIPS могут поддерживать как «little-endian» так и «big-endian» архитектуру памяти. Для определенности рассмотрим «little-endian» архитектуру, которая и моделируется в симуляторе MARS. Схема расположения

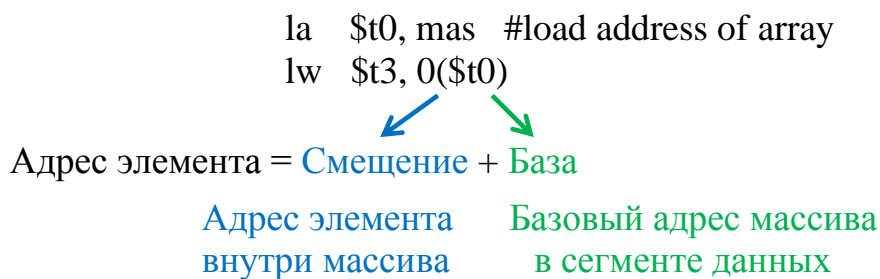


и обращения к элементу массива будет аналогична порядку использования памяти в Intel. Формула для вычисления адреса элемента в памяти будет та же:

$$\text{Адрес эл-та} = (\text{адрес нач. массива}) + (\text{№эл-та}) \times (\text{размер эл-та в байтах})$$

Начальный адрес массива и его размер (число элементов) заносятся в целочисленные регистры (можно использовать любой свободный), и на каждой итерации указатель изменяется на длину элемента в байтах (см. пример ниже).

Обращение к элементу массива, к оперативной памяти в MIPS организовано единственно возможным способом – с помощью базовой адресации со смещением:



Немного другая форма, но также базовая адресация:



## 2.2. Организация цикла (for, while)

В Ассемблере MIPS нет специальных инструкций для организации цикла. Для перебора элементов массива в цикле требуется продумать самостоятельно вид цикла, регистр-счетчик количества повторений цикла, условие окончания, регистр для хранения флага окончания и временного индикатора для сравнения с индикатором. Проверка условий выполняется операторами «если-то-иначе», организованными командами условного перехода. (bc1f, bc1f, bc1t, bc1t, beq, bgez, bgezal, bgtz, blez, bltz, bltzal, bne) или безусловного перехода (j, jal, jalr, jalr, jr).

```

.data
mas:   .word  -1 -2 -3 -4 1 2 0 4 2000 4096 65535 0    # "array" of 12 words separated by spaces
size:  .word  12          # size of "array"
head:  .asciiz "\n Null element amount: "

.text
    la  $t0, mas          # load address of array – initial index [i]
    lw  $t5, size         # load array size in the register

loop:  lw  $t3, 0($t0)     # get value from array mas[i] into register $t3
       bne $0, $t3, met1  # branch for met1 if register $t3 not equal '0' mas element is not 'zero'
       addi $t4, $t4, 1   # increment Null counter
met1:  addi $t5, $t5, -1   # decrement loop counter
       addi $t0, $t0, 4   # increment index [i] (add the element length, 4 bytes for 'word' type)
       bgtz $t5, loop     # repeat if not finished yet (Branch to loop label's address if $t5 is greater than zero)

    la  $a0, head        # load address of print heading
    li  $v0, 4           # specify Print String service
    syscall              # print heading
    add $a0, $t4, $zero   # load value for print into argument register $a0, using pseudo-op
    li  $v0, 1           # specify Print Integer service
    syscall              # print Null counter

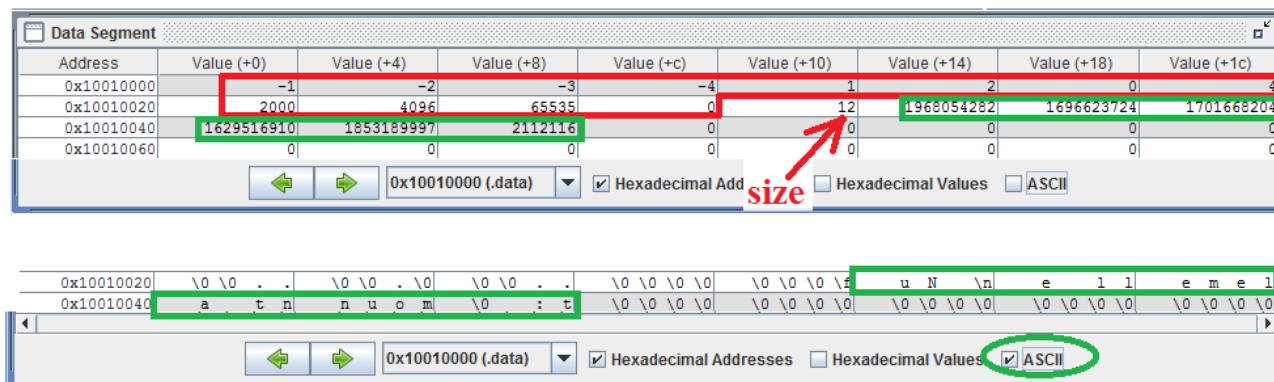
```

### 2.3. Использование двумерных массивов (матриц)

Для использования двумерных массивов следует самостоятельно продумать способ адресации элементов с помощью запрограммированной формулы вычисления № строки и № столбца.

Выше приведен пример программы подсчета количества нулевых элементов в одномерном массиве.

Согласно вышеприведенной программе в памяти последовательно расположены 12 четырехбайтовых элементов массива *mas* (среди которых есть два нулевых) – выделено на рис. ниже красными линиями, затем переменная *size*, содержащая размер массива (12) – показано красной стрелкой, затем ASCII-коды символов выводимого сообщения *head* (которые можно увидеть, перейдя в соответствующий режим отображения) – выделено зелеными линиями. Следует помнить об особенности пословного отображения архитектуры «little-endian», где слова показаны в привычном виде (со старших байтов), а потому в перевернутом виде по отношению к хранимой в памяти информации.



Далее следуют три инструкции загрузки:

- 1) адреса массива,
- 2) адреса переменной-размера массива,
- 3) значения переменной-размера массива.

Затем начинается цикл просмотра элементов (метка *loop*), в котором

- загружается очередной элемент массива *mas[i]*,
- сравнивается с нулём и при необходимости увеличивается счетчик нулей,
- уменьшается счетчик цикла – количество оставшихся итераций цикла,

- увеличивается адрес элемента на его дину (4 Байта),
- проверяется условие окончания цикла (счетчик цикла равен нулю).

Далее выполняются два системных вызова

- для вывода строки-сообщения,
- для вывода переменной – счетчика нулей.

Результат работы программы следующий:

The screenshot shows a window titled "Mars Messages" with a "Run I/O" button. The output text is:
 

```
Null element amount: 2
-- program is finished running (dropped off bottom) --
```

 There is also a "Clear" button on the left side of the window.

### 3 ЗАДАНИЕ

*Работа может быть выполнена группой из двух студентов. Правила оценивания прежние.*

1. Создать две программы согласно варианту: для Intel и MIPS, которая
  - выполняет заданные действия,
  - выводит на экран сообщения о результатах/ошибках.
2. Откомпилировать программу и запустить на исполнение (в случае безошибочной компиляции, иначе исправить синтаксические ошибки и провести повторную компиляцию и запуск).
3. Отладить программу либо в среде OllyDbg или др. отладчиках для Intel, либо в пакете MARS (отладчик может быть выбран студентами самостоятельно), проследить изменения в регистрах и ОП.
4. Ответить на вопросы для самопроверки.

### 4 ВАРИАНТЫ ЗАДАНИЙ

№ вар	Задание (целочисленные операции)
1.	Найти минимальный и максимальный элемент массива А и запомнить их номера и адреса в переменных в памяти А_min, А_max. Вывести

	переменные, и предусмотреть случай (вывести сообщение), когда имеются все/несколько минимальных или максимальных элементов.
2.	Для заданного массива $A$ посчитать и запомнить количество нулевых/отрицательных/положительных элементов. Вывести сообщения с этими значениями и предусмотреть случай (вывести сообщение), когда какие-то типы элементов отсутствуют.
3.	Найти элементы массива $A$ , содержащие 2 единицы в двоичном представлении и запомнить их номера и адреса в новых одномерных массивах $V\_nom$ и $V\_adr$ , предусмотреть случай, когда таких элементов нет. Вывести сообщение о количестве таких элементов или об их отсутствии.
4.	Для заданного массива $A$ составить новый массив $B$ , каждый элемент которого $B[i]$ соответствует № первого слева (старшего) нулевого разряда элемента $A[i]$ . Предусмотреть случай отсутствия нулевых разрядов элемента. Вывести сообщение с числом таких элементов или штатном режиме – весь массив $B$ заполнен.
5.	Для заданного массива $A$ составить новый массив $B$ , каждый элемент которого $B[i]$ соответствует № первого справа (младшего) единичного разряда элемента $A[i]$ . Предусмотреть случай отсутствия единичных разрядов элемента. Вывести сообщение с числом таких элементов или штатном режиме – весь массив $B$ заполнен.
6.	Для заданного массива $A$ составить новый массив $B$ , каждый элемент которого $B[i]$ равен сумме единиц в двоичном представлении элемента $A[i]$ . Посчитать общее количество единиц. Предусмотреть случай отсутствия единичных разрядов элементов. Вывести сообщение с общим количеством единиц или об их отсутствии.
7.	Вычислить сумму всех отрицательных, положительных элементов массива $A$ , число нулевых элементов, число четных и нечетных

	элементов и запомнить эти 5 значений в указанном порядке в одномерном массиве В. Вывести сообщения о полученных результатах, предусмотреть случай, когда некоторые виды элементов отсутствуют.
8.	Вычислить сумму всех нулевых и единичных битов в двоичном представлении элементов массива А, запомнить эти значения в памяти и вывести эти значения или сообщение об отсутствии некоторых видов битов.
9.	Для заданного массива А рассчитать разницу между максимальным и минимальным элементом и вывести это число. Предусмотреть случай их отсутствия (когда все элементы массива равны) и вывести об этом сообщение.
10.	Вычислить сумму всех элементов массива А, больше заданной переменной. Вывести сообщения о полученных результатах, предусмотреть случай, когда такие элементы отсутствуют.
11.	Для заданного массива А составить новый одномерный массив символьных строк В. Первым символом строки В[i] будет символ «Р» (positive), если элемент А[i] положительный и «N», если отрицательный. Вторым символом строки В[i] будет «-», третьим символом строки В[i] будет «Z», если элемент А[i] нулевой. Вывести сообщения об отсутствии одного из типов элементов и числе положительных элементов.

## 5 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как обратиться к элементу массива в программе на Ассемблере?
2. Как указывается адрес элемента массива/ячейки памяти?
3. Какими бывают режимы адресации элементов массивов? Приведите пример из вашей программы.
4. Как располагаются элементы массива в памяти компьютера? Приведите пример из вашей программы.

5. Как рассчитать адрес элемента с произвольным номером?
6. Как организовать цикл для перебора всех элементов массива?
7. Как организуется счётчик циклов?
8. Как изменяется номер и адрес элемента массива в цикле?
9. Как организовать вывод числовых целочисленных значений для MIPS?
10. Как работает системный вызов syscall? Какие у него есть сервисы?
11. Расскажите об изменениях в регистрах процессора/ячейках памяти по ходу выполнения вашей программы.

## **6 СПИСОК ЛИТЕРАТУРЫ**

1. Нарваха Р. Введение в крэкинг с нуля, используя OllyDbg. (<http://pro.dtn.ru/cr.html>).
2. Intel 64 and IA32-Architectures Software Developers Manual (с сайта intel.com)
3. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. – СПб.: Издательство «Питер», 2000.
4. Intel 64 and IA32 Architectures Software Developers Manual.
5. Юров В. Ассемблер. – СПб.: Издательство «Питер», 2002.
6. <https://courses.missouristate.edu/KenVollmar/MARS/download.htm>
7. [https://github.com/MIPSfpga/schoolMIPS/tree/00\\_simple/scripts/bin](https://github.com/MIPSfpga/schoolMIPS/tree/00_simple/scripts/bin)
8. Хэррис, Д. М. Цифровая схемотехника и архитектура компьютера, 2017/2019г.
9. <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>.
10. Сайт «Учебный курс: Методы | Регистры MIPS»
11. <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>.
12. Сайт «Учебный курс: Методы | Регистры MIPS»

## ГЛАВА 4. РАБОТА С ДЕЙСТВИТЕЛЬНЫМИ ЧИСЛАМИ И РЕГИСТРАМИ БЛОКА FPU. Практическая работа №4

### 1. ТЕОРИЯ

#### 1.1. Формат чисел с плавающей запятой (ЧПЗ или FP)

Дробные (действительные) числа представлены в компьютере в двоичном коде в формате ЧПЗ – число с плавающей запятой (или FP – float point) [<https://habr.com/post/112953/>].

#### Пример.

Представить десятичное число  $52,161_{10}$  в двоичной системе счисления.

#### Решение

1) сначала преобразуем целую часть числа:

$$52/2 = 26 \text{ (остаток } 0, \text{ значит цифра } a_0=0) ;$$

$$26/2 = 13 \text{ (остаток } 0 \text{ – цифра } a_1) ;$$

$$13/2 = 6 \text{ (остаток } 1 \text{ – цифра } a_2) ;$$

$$6/2 = 3 \text{ (остаток } 0 \text{ – цифра } a_3) ;$$

$$3/2 = 1 \text{ (остаток } 1 \text{ – цифра } a_4) ;$$

$$1/2 = 0 \text{ (остаток } 1 \text{ – цифра } a_5) .$$

2) затем преобразуем дробную часть числа:

$$0,161 \cdot 2 = 0,322 \text{ (целая часть равна } 0, \text{ значит цифра } a_{-1}=0) ;$$

$$0,322 \cdot 2 = 0,644 \text{ (цифра } a_{-2}=0) ;$$

$$0,644 \cdot 2 = 1,288 \text{ (цифра } a_{-3}=1) ;$$

$$0,288 \cdot 2 = 0,576 \text{ (цифра } a_{-4}=0) ;$$

$$0,576 \cdot 2 = 1,152 \text{ (цифра } a_{-5}=1) ;$$

$$0,152 \cdot 2 = 0,304 \text{ (цифра } a_{-6}=0) ;$$

$$0,304 \cdot 2 = 0,608 \text{ (цифра } a_{-7}=0) ;$$

$$0,608 \cdot 2 = 1,216 \text{ (цифра } a_{-8}=1) \dots$$

Таким образом,  $52,161_{10} \approx 110100,00101001_2$ .

Однако если мы попробуем проверить вычисления и сделаем обратный перевод двоичного числа  $110100,00101001_2$  в десятичное, то получим

Иванова Е.М. Учебное пособие по разделу «Вычислительные системы»



приближённое число  $52,16015625_{10}$ . Значит, погрешность перевода (в десятичном выражении) составила 0.00084375. Это происходит потому, что преобразование дробной части числа умножением – процесс, в принципе, бесконечный. И когда мы останавливаемся принудительно, то отбрасываем значащие цифры числа, которые могли бы получиться при дальнейших вычислениях.

При машинном представлении дробей существует правило: умножение, используемое для преобразования дробной части числа, продолжается до тех пор, пока не наступит одно из следующих событий:

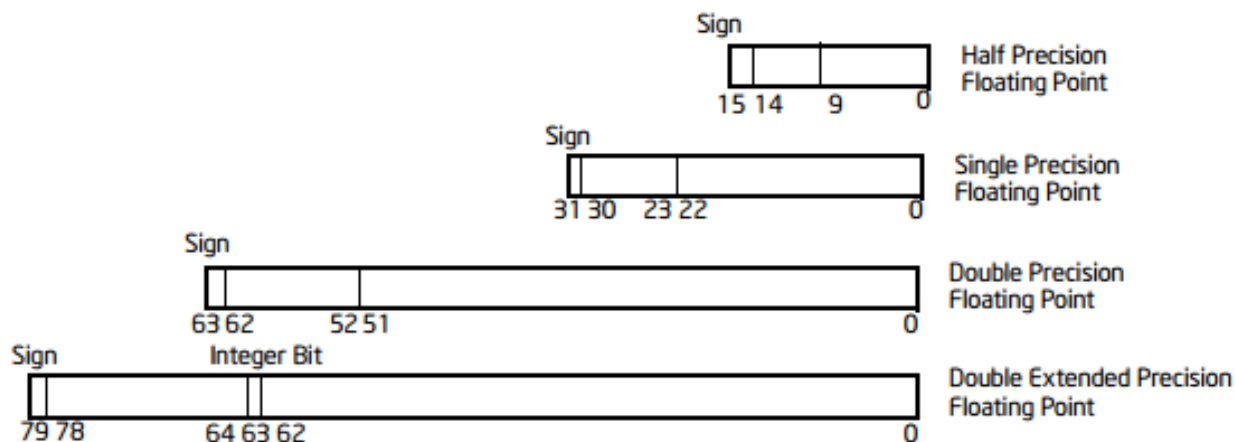
1. достигнута необходимая точность представления (набрано нужное количество разрядов: 23, 52, 64);
2. при очередном умножении получается точное значение единицы;
3. процесс «зацикливается».

Для работы с дробями в компьютере используется формат ЧПЗ. Большинство производителей процессоров, в том числе Intel и AMD, придерживаются международного стандарта IEEE 754–2008, регламентирующего способы хранения и методы обработки таких чисел – в нормализованной форме в двоичной системе счисления.

Тип данных	Размер (байт)	Точность	Разрядность (бит)			
			<i>s</i> знак	<i>p</i> порядок	<i>m</i> мантисса	всего
binary16	2	половинная	1	5	10	16
binary32 (float)	4	одинарная	1	8	23	32
binary64 (double)	8	двойная	1	11	52	64
Не стандартизованный формат Intel	10	расширенная	1	15	64	80
binary128 (long double)	16	учетверённая	1	15	112	128

Согласно этому стандарту процессоры Intel используют числа с плавающей точкой трёх типов с половинной (используется редко), одинарной, двойной и

расширенной точностью: binary16, binary32, binary64, binary128, (некоторые соответствующие типы в языке C – float, double, long double).



В формате с плавающей точкой число в нормализованной форме для хранения в ВС вне зависимости от типа представляется в виде:

$$A_2 = (-1)^Z \cdot M \cdot 2^X$$

где  $Z$  – знак числа,  $M$  – мантисса, записанная с основанием 2,  $X$  – характеристика (экспонента) числа.

Знак числа 1 разряд	$X$ – характеристика ( $p$ разрядов)	$M$ – мантисса ( $m$ разрядов)
------------------------	---	-----------------------------------

$M = a_{-1} a_{-2} \dots a_{-m}$  – дробная часть мантиссы, представленная в прямом коде.

Причём разряд  $a_0$  (целая часть =1) может

- храниться в числе (в формате long double)
- не храниться в числе (в формате float или double)

$Z$  – знак числа, для положительных чисел равен нулю (+число  $\rightarrow Z=0$ ), для отрицательных чисел равен единице (–число  $\rightarrow Z=1$ ).

$X$  – характеристика числа (целая положительная), которая вычисляется как смещенный порядок: она равна истинному порядку  $P$  (может быть как положительный, так и отрицательный), увеличенному на значение смещения.

Значение смещения для трех разных форматов равно

- $2^7 - 1 = 127$  (8 бит для float),
- $2^{10} - 1 = 1023$  (11 бит для double) и
- $2^{14} - 1 = 16383$  (15 бит для long double).

**Пример 1.**

Возьмем число  $-247,375$ .

1. В двоичном представлении в прямом коде число будет выглядеть так:

$$-11110111,011.$$

2. Нормализуем число, перенося запятую на 7 знаков влево. Получаем

$$-1,1110111011 \cdot 2^7,$$

Соответственно,  $7=+111$  – истинный порядок числа.

3. Найдем **характеристику** (смещенный порядок) для разных типов чисел:

float – чисел одинарной точности (8 разрядов)  $7+127=134=10000110$ ,

double – чисел двойной точности (11 разрядов)  $7+1023=10000000110$ ,

long double – расширенной точности (15 разрядов)  $7+16383=16390=100000000000110$ .

4. Найдем **мантиссу** для разных типов чисел.

Для типа

**float** – чисел одинарной точности длина мантиссы должна быть 23 разряда

$$1, \underbrace{11101110110000000000000}_{23}$$

**double** – чисел двойной точности длина мантиссы должна быть 52 разряда

$$1, \underbrace{111011101100000000000000 \dots 00000000000000000000}_{52}$$

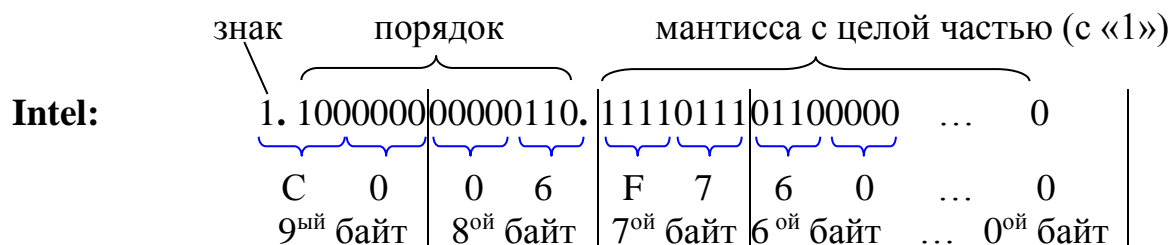
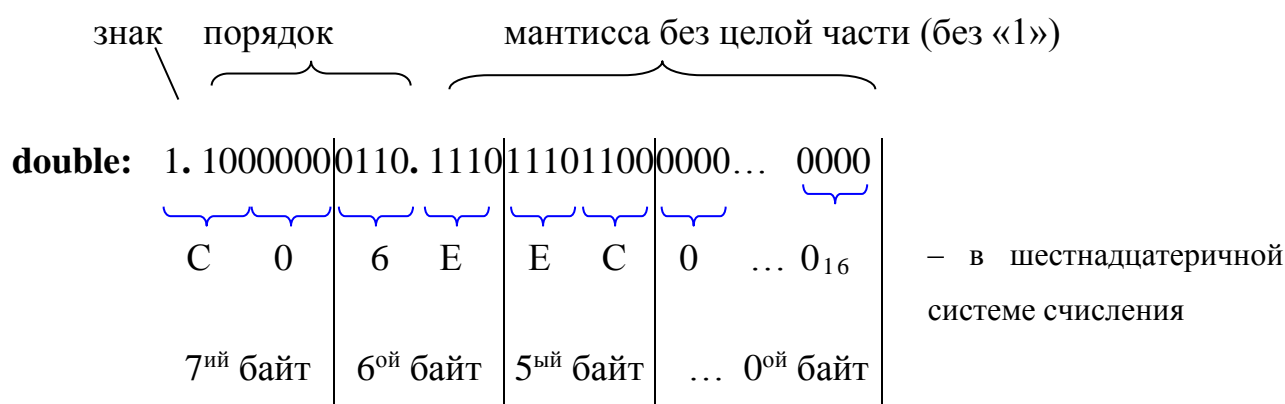
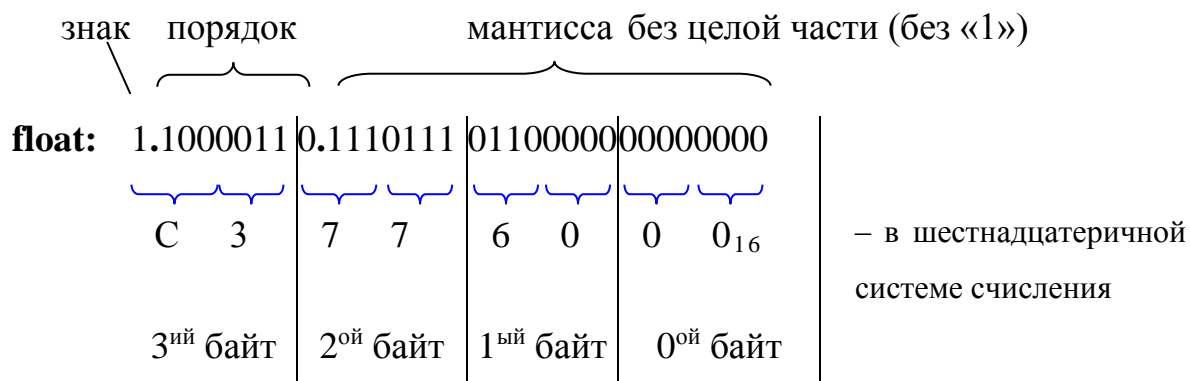
Обратите внимание, что в форматах float и double единица целой части не хранится в самой мантиссе.

**Intel** – расширенной длина мантиссы должна быть 64 разряда

$$1, \underbrace{11101110110000000000000000000000 \dots 00000000000000000000}_{64}$$

Обратите внимание, что в формате Intel единица целой части хранится в самой мантиссе в качестве её первого числового разряда.

## Окончательный результат: <знак><порядок><мантисса>

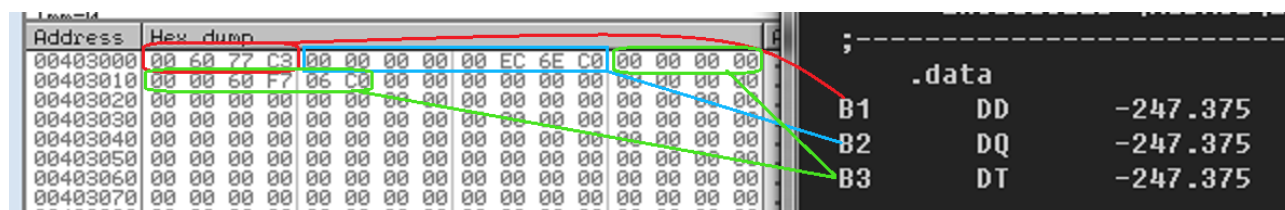
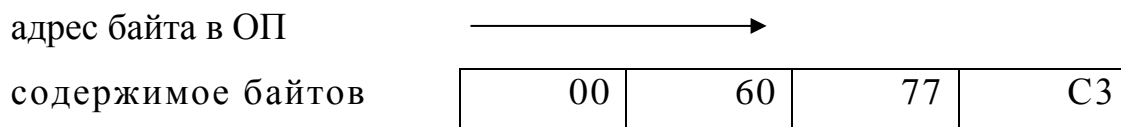


Итак, в памяти ВС десятичное дробное число  $-247,375$  может соответствовать в зависимости от точности представления различным двоичным кодам (для краткости они показаны в шестнадцатеричной системе счисления по аналогии с результатами любого отладчика при просмотре содержимого ячеек памяти и регистров процессора):

float:	4 байта	C3 77 60 00 <sub>16</sub>
double:	8 байт	C0 6E EC 00 00 00 00 00 <sub>16</sub>
Intel:	10 байт	C0 06 F7 60 00 00 00 00 00 00 <sub>16</sub>

## 1.2. Работа с ЧПЗ в процессоре Intel

Поскольку Intel x86-совместимые компьютеры имеют архитектуру little-endian (очень интересно происхождение термина [<http://ru.wikipedia.org>]), в памяти число будет выглядеть как перевернутая последовательность байт, согласно правилу: младший байт по младшему адресу. Например, при размещении числа  $-247,375$  в формате float имеем последовательность байт:



В этом можно убедиться, если скомпилировать и выполнить небольшую программу на языке Си:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    float f = -0.1015625f;
    unsigned char *p = (unsigned char *) &f;
    printf("%02x %02x %02x %02x\n",
        (unsigned int) p[0],
        (unsigned int) p[1],
        (unsigned int) p[2],
        (unsigned int) p[3]);
    return 0;
}
```

Эта программа выдаст ожидаемый результат: 00 60 77 C3.

*Для других типов (double, long double) можно слегка подправить программу и также увидеть хранимые в памяти коды для любого заданного дробного числа.*

### **Пример 2.**

Допустим в ячейках памяти расположено действительное число В. Мы знаем, что оно 4-байтное и видим его 16-ричное представление

00 00 F2 C2

Согласно архитектуре little-endian привычное представление числа со старших разрядов слева направо наше число выглядит как  $C2F20000_{16}$ . Зная, что это число в формате с плавающей запятой, определим реальное значение числа в десятичной системе. Количество байтов, занимаемых числом – 4, значит число – типа float.

1. Переведем число в двоичную систему:

1100 0010 1111 0010 0000 0000 0000 0000.

2. Старший знак числа «1», значит число отрицательное.

3. Характеристика в типе float занимает 8 бит, начиная со второго.

Получаем  $10000101=133$ .

4. Истинное значение порядка  $133 - 127 = 6 (2^6)$ .

5. Мантисса равна :  $(1,)111 0010 0000 0000 0000 0000$ . Нормализованное двоичное представление числа имеет вид:

$- 1,111001 \cdot 2^6$ .

6. Чтобы получить более привычное представление (без порядка) смещаем мантиссу на 6 знаков вправо, получаем

$- 1 111 001,0_2$ .

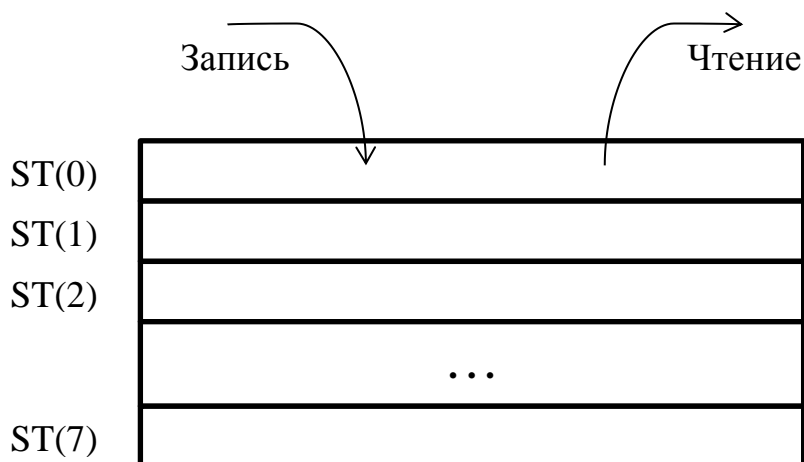
Что соответствует десятичному числу:

$-121,0$ .

Код любой ячейки памяти может рассматриваться как код дробного числа формата ЧПЗ, если к нему применяется команда обработки дробного числа. Такие команды начинаются с буквы F (FADD, FABS, FLD...).

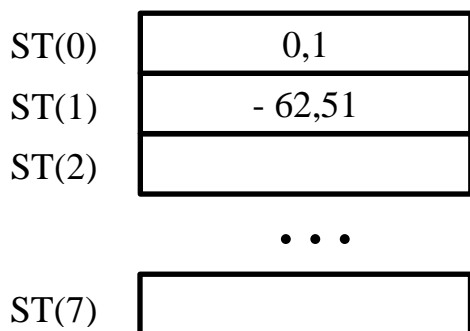
В блоке сопроцессора предусмотрены регистры для хранения дробных значений (ST(0), ... ST(7)). Обратиться к такому регистру можно по имени (FADD ST(0),ST(1)), как к другим регистрам ЦП.

Поместить дробное значение в регистр можно командой FLD, а прочитать значение можно из вершины стека ST командой FST. ST0÷ST7 образуют аппаратный стек регистров FPU

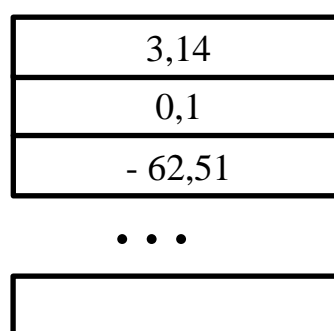


При записи нового значения все имеющиеся в стеке величины «как бы смещаются вглубь» стека. Например, команда FLDPI загружает в стек константу – число  $\pi$ .

Состояние стека ДО записи  $\pi$

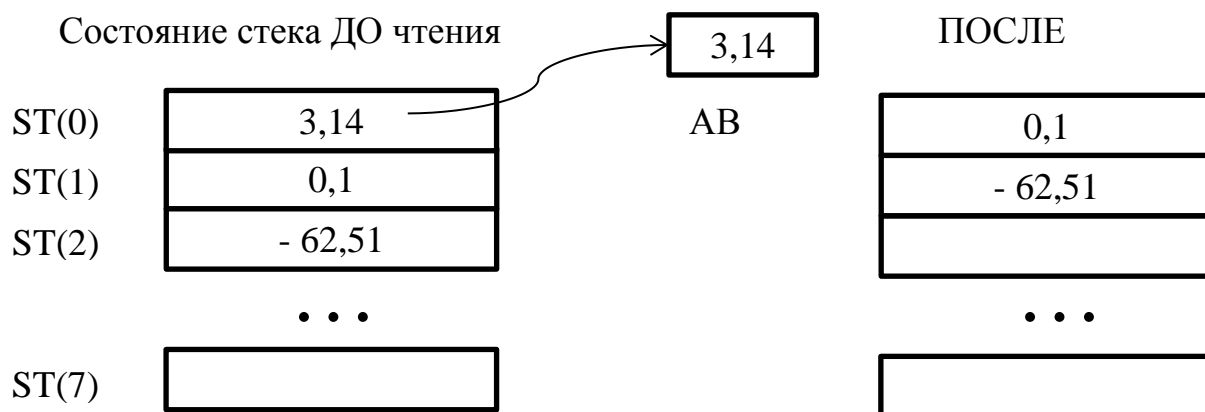


ПОСЛЕ

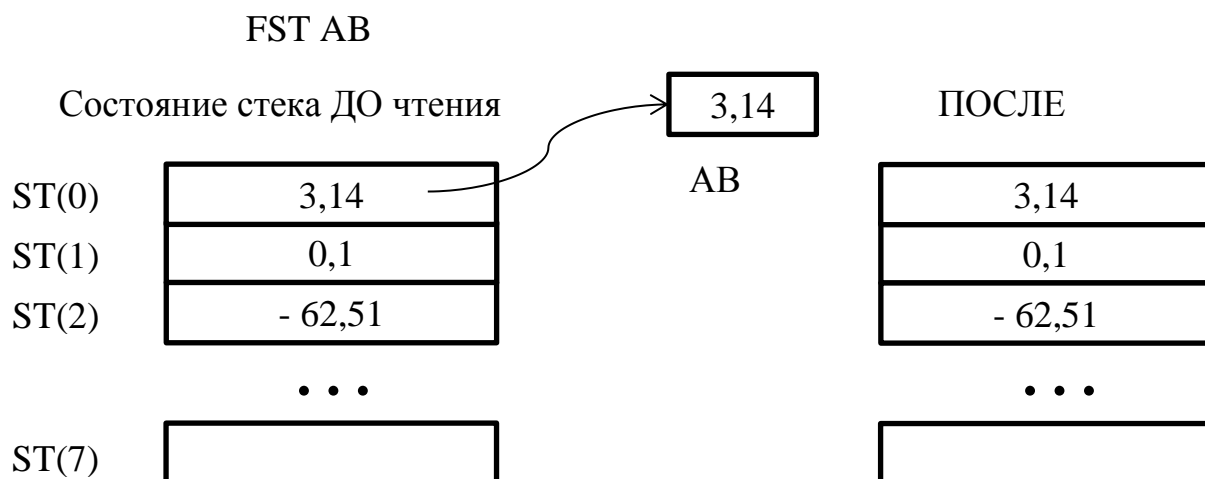


Чтобы стек не переполнялся можно воспользоваться командой чтения из стека с выталкиванием из вершины стека значения:

FSTP AB ; занесение числа из регистра ST(0) в переменную AB  
; с изменением указателя стека регистров FPU.



Если значение в стеке требуется для дальнейших вычислений, то его можно не выталкивать. Чтение из стека без выталкивания сохранит все значения в стеке неизменными:



Команды обработки чисел формата ЧПЗ можно посмотреть в [1-3].

На самом деле имена регистров ST(0) ... ST(7) не привязаны жестко к конкретному физическому регистру. У стека FPU есть указатель, который находится в 13-12-11 разрядах служебного регистра SWR (или поле SWR.TOP). В OllyDbg этот регистр называется FST.

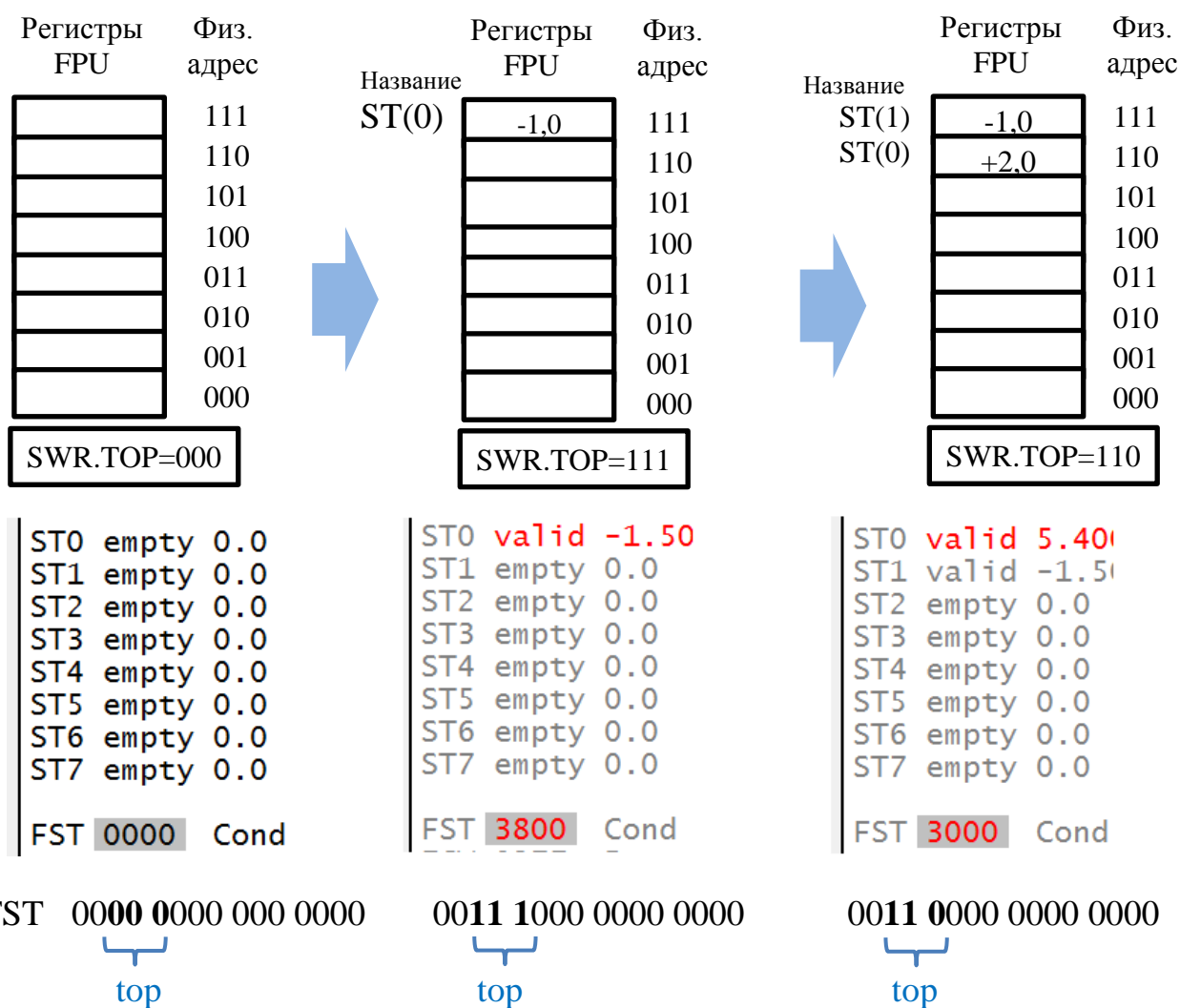
Вершина стека будет условно называться ST(0) или просто ST. То, что лежит глубже в стеке будет называться ST(1), ST(2) и т.д. Но номер в названии регистра не совпадает с его физическим адресом в блоке регистров FPU (а именно этот адрес храниться в SWR.TOP). Изначально разряды SWR.TOP содержат «000». Стек пуст и вершина отсутствует.

Вершина стека всегда должна находиться в ST(0), который иногда обозначается просто как ST. Поэтому при добавлении в стек новой записи



происходит переименование регистров и вершины. Перед записью в стек FPU адрес вершины стека уменьшится на 1, т.к. любой стек заполняется в сторону уменьшения адресов.

Когда стек пуст, т.е. адрес вершины уже был «000», то перед занесением первого значения (-1,0) указатель сначала станет максимальным № регистра FPU, т.е. 111. После очередной записи в стек FPU (например, числа +2,0) адрес вершины будет 110 – это и будет ST(0), а прежняя вершина переименуется в ST(1) – т.е. как бы опуститься глубже в стек.



Далее приведён пример программы, которая вычисляет сумму двух дробных чисел и заносит результат в память с освобождением стека.

.386

.model flat, stdcall

option casemap :none ; case sensitive

Иванова Е.М. Учебное пособие по разделу «Вычислительные системы»

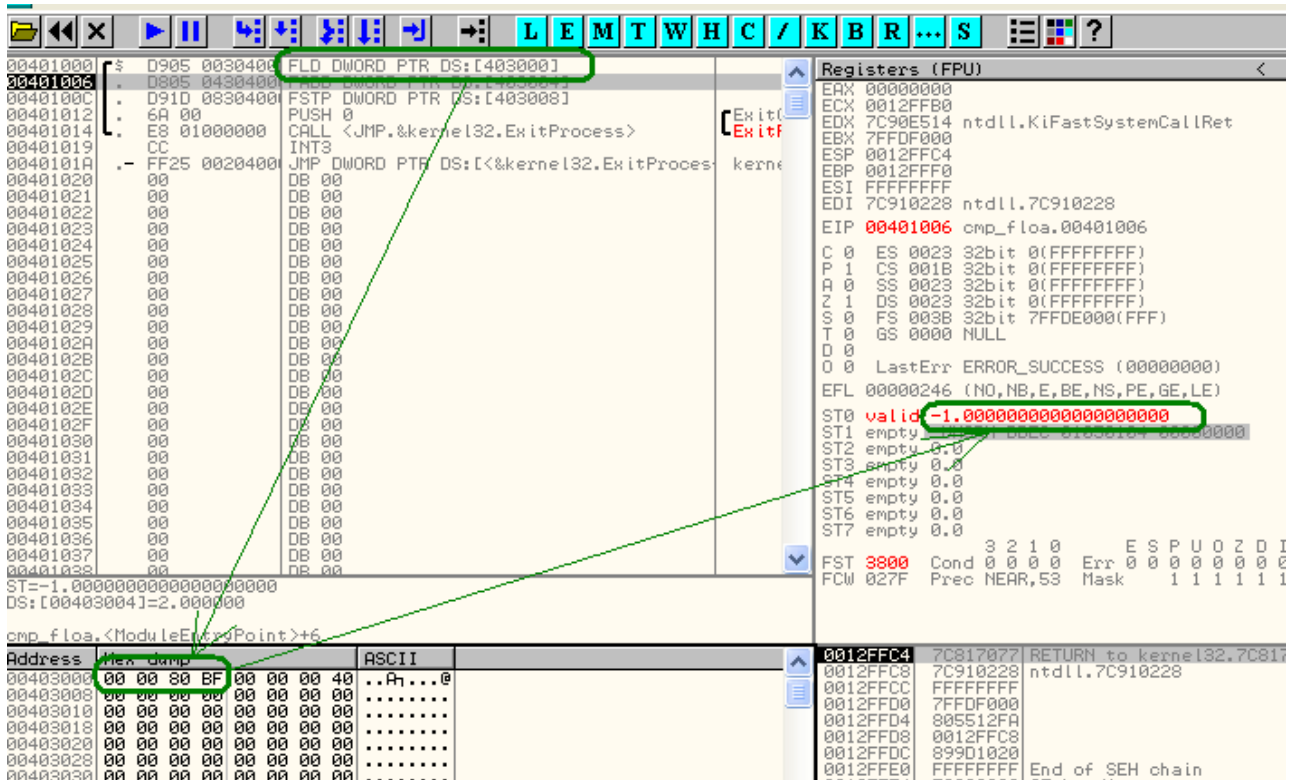
```

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
include \masm32\include\fpu.inc ;
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\fpu.lib ;

;-----
.data
A DD -1.0
B DD +2.0
sumAB DT 0

;-----
.code
start:
FLD A
FADD B
FSTP sumAB
invoke ExitProcess, NULL
end start ;Конец программы

```



Если посмотреть на выполнение этой программы в OllyDbg, то можно увидеть, что после первой команды число  $A = -1,0$  будет помещено в стек (в регистр ST(0)). Причём, если его размер меньше регистра (как в нашем случае), то число будет расширено до формата long double.

Кроме того, каждое число в регистре ST снабжено тегом, поясняющем состоянии регистра:

```

ST0 zero 0.0
ST1 valid 1.00000000000000000000
ST2 zero -0.0
ST3 bad -??? FFFF 00000000 00000000
ST4 bad +INF 7FFF 80000000 00000000
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

```

- empty – пустой,
- valid – достоверно,
- zero – ±ноль,
- bad – неопределённость.

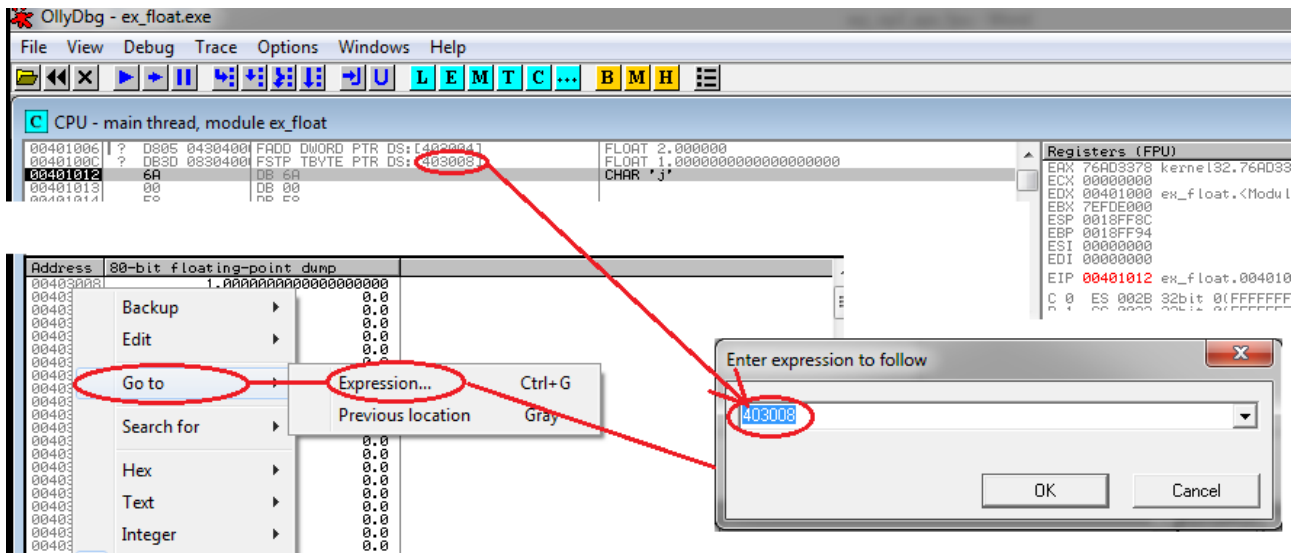
Далее после выполнения двух других команд сумма  $A+B=+1,0$  будет запомнена в десяти-байтовой переменной sumAB в перевернутом виде:

0-й байт по адресу sumAB,

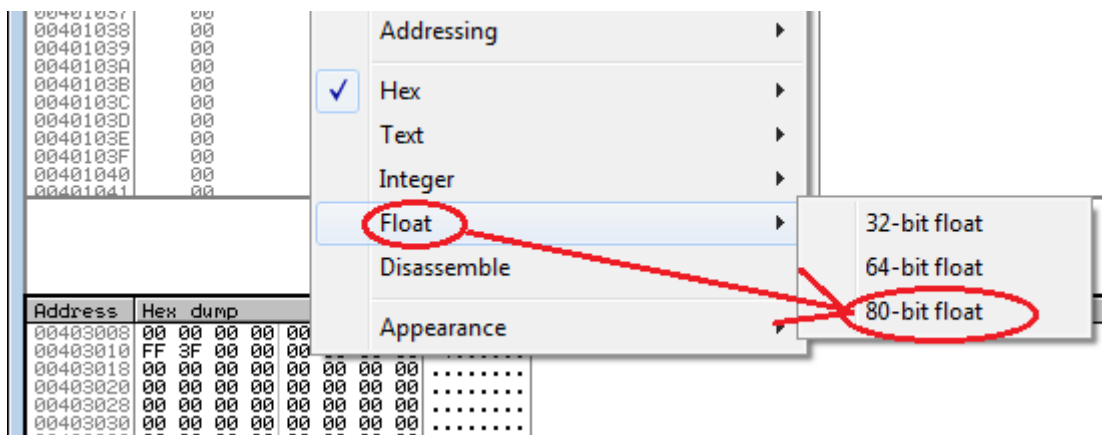
1-й байт по адресу sumAB+1, ... 9-й байт по адресу sumAB+9.

The screenshot displays the OllyDbg interface. The CPU registers window shows ST0 as 'valid 1.00000000000000000000'. A 'Modify ST0' dialog box is open, allowing the user to change the value of the ST0 register. The 'Float' field contains '1.00000000000000000000' and the 'Hex' field contains '3FFF 80000000 00000000'. The 'Modify associated tag' checkbox is checked. The background shows the CPU registers window with ST0 set to 'valid 1.00000000000000000000' and the memory dump window showing the stack content.

При других настройках можно увидеть непосредственное десятичное значение переменной в памяти. Для этого нужно перейти по адресу расположения переменной в ОП (dump): кликнуть правой кнопкой мыши на адресном поле дампа памяти и в выпадающем меню выбрать опцию «Go to» и далее опцию «Expression». В открывшемся окне набрать адрес переменной (этот адрес будет отражён в команде, где переменная используется) и нажать «OK».



Дамп памяти будет отображаться начиная с указанного адреса. Далее повторить процедуру настройки отображения дампа памяти, но выбрать опцию «Float» и далее указать требуемый формат (32-64-80).



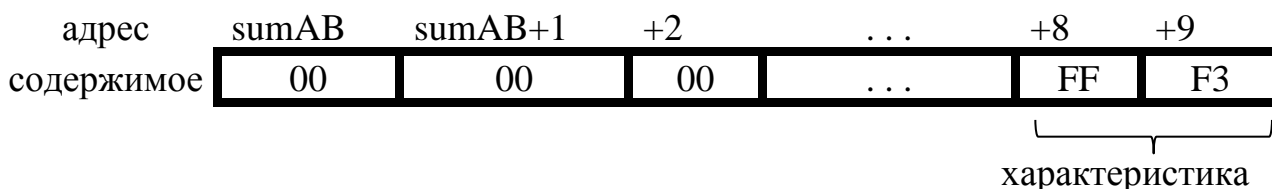
После всех указанных настроек вы сможете увидеть десятичное представления дробного результата.

Address	80-bit floating-point dump
00403008	1.00000000000000000000000000000000
00403012	0.0
0040301C	0.0
00403026	0.0
00403030	0.0
0040303A	0.0

Для умножения или деления на степень двойки (2,4,8,16,32, ...) в нормализованном числе формата ЧПЗ можно изменить только характеристику, которая, собственно, и показывает степень двойки. Для деления на  $2^N$  надо из характеристики вычесть N, а для умножения – прибавить N. Помня формат

числа, и как оно храниться в памяти, можно вычлениТЬ байты характеристики и изменить их.

```
fld    A           ;занесение в стек регистров числа A (-1.0)
fadd   B           ;сложение A с B (-1.0+2.0)
fstp   sumAB       ;запоминание суммы в переменной sumAB (+1.0)
lea    ESI,sumAB+8 ;загрузка в указатель ESI адреса начала
                    ; характеристики десяти-байтового числа sumAB
inc    [ESI]       ;увеличение на 1 характеристики = sumAB×2
fld    sumAB       ;занесение в стек регистров увеличенного числа (+2.0)
```



Можно также воспользоваться командами выделения мантииссы и порядка и масштабирования:

```
.data
A      DD  -1.4
B      DD  +2.0
sumAB  DT   0
;-----
.code
start:
    FLD  A      ;занесение в вершину стека ST0 числа A из памяти
    FXTRACT    ;выделение мантииссы (в ST0) и порядка (в ST1)
    FXCH       ;обмен содержимого регистров ST0 и ST1
    FADD  B     ;сложение ST0 с числом B
    FXCH       ;обмен содержимого регистров ST0 и ST1
    FSCALE    ;вычисление ST0= ST0×2ST1
    FSTP sumAB ;сохранение в памяти результата
    ....
```

### 1.3. Вывод ЧПЗ в окно

Для вывода на экран результата вычислений в формате FP можно использовать функцию `FpuFLtoA` [7]. Данная функция не является API-функцией, а является внутренней функцией MASM32 и входит в спец. Библиотеку `FPU.lib`. Подробное описание функции можно найти в справке MASM32 из каталога `masm32\help\fpuhelp.chm` (Available Functions, example).

`invoke FpuFLtoA, addr S, 5, addr sResult, SRC1_REAL or SRC2_DIMM or STR_SCI`

**addr S** - адрес 80-разрядного дробного числа для вывода

**8** - количество разрядов после запятой

**addr sResult** – адрес буфера (строки), куда будут записаны преобразованные в строку разряды дробного числа

далее **флаги**, управляющие работой функции, объединённые по «ИЛИ»

значения флагов:

**SRC1\_FPU** – первый параметр функции не используется

**SRC1\_REAL** – первый параметр функции д.б. 80-разрядным дробным числом

**SRC2\_DMEM** – второй параметр функции д.б. адресом 32-разрядного целого беззнакового числа

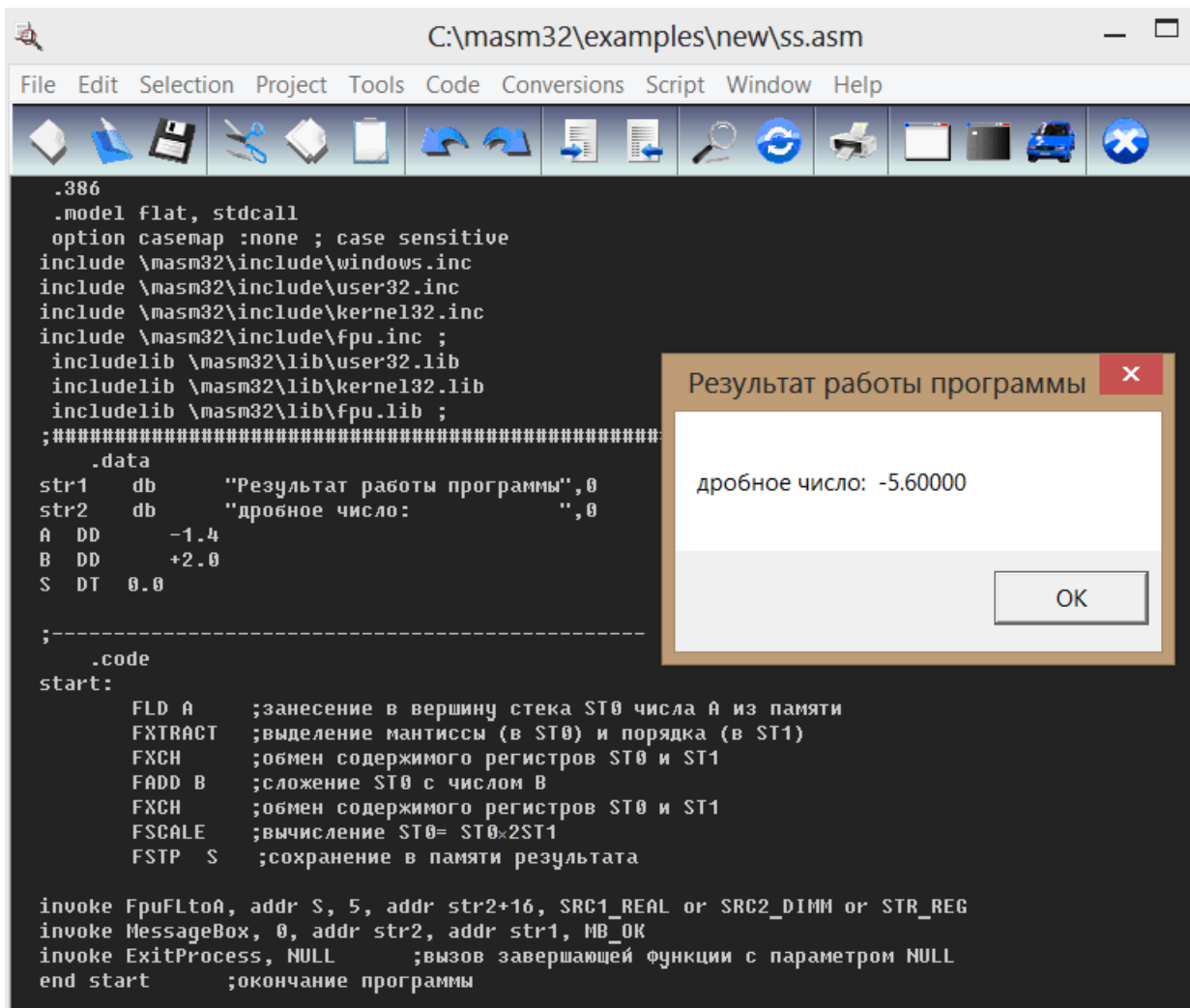
**SRC2\_DIMM** – второй параметр функции д.б. непосредственно заданным (константой) 32-разрядным целым беззнаковым числом

(как в примере «`FpuFLtoA, addr S, 5, addr sResult, SRC1_REAL`»)

**STR\_REG** – число выводится в десятичном формате (–3,00000)

**STR\_SCI** – число выводится в научном формате (–3,00000E+0000)

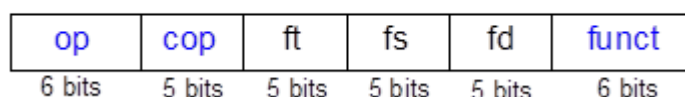
Для рассмотрено выше примера вывод выглядит так:



### 1.4. Работа с ЧПЗ в процессоре MIPS

Как и в процессоре Intel в MIPS существуют отдельные инструкции для обработки ЧПЗ, однако их гораздо меньше. Для операций с плавающей точкой в MIPS используется опциональный сопроцессор (Coprocessor 1), который содержит 32 32-разрядных регистра для хранения операндов в формате с плавающей точкой одинарной точности (\$f0-\$f31). Для хранения операндов с двойной точностью (64 разряда) используется пара таких регистров, например, \$f0 и \$f1, \$f2 и \$f3 и т.д., поэтому для операций с числами двойной точности доступны только 16 четных регистров: \$f0, \$f2, \$f4 и т.д. Для ЧПЗ используются инструкции F-типа.

#### F-Type



Код операции (поле «op») у всех команд с плавающей точкой равен 17

Registers		
	Coproc 1	Coproc 0
Name	Float	Double
\$f0	0x00000000	0x41b0cccc00000000
\$f1	0x41b0cccc	
\$f2	0x4206cccc	0x000000004206cccc
\$f3	0x00000000	
\$f4	0x00000000	0x0000000000000000
\$f5	0x00000000	
\$f6	0x00000000	0x0000000000000000
\$f7	0x00000000	
\$f8	0x00000000	0x0000000000000000
\$f9	0x00000000	
\$f10	0x00000000	0x0000000000000000
\$f11	0x00000000	
\$f12	0x425f3334	0x00000000425f3334
\$f13	0x00000000	
\$f14	0x00000000	0x0000000000000000
\$f15	0x00000000	
\$f16	0x00000000	0x0000000000000000
\$f17	0x00000000	
\$f18	0x00000000	0x0000000000000000
\$f19	0x00000000	
\$f20	0x00000000	0x0000000000000000
\$f21	0x00000000	
\$f22	0x00000000	0x0000000000000000
\$f23	0x00000000	
\$f24	0x00000000	0x0000000000000000
\$f25	0x00000000	
\$f26	0x00000000	0x0000000000000000
\$f27	0x00000000	
\$f28	0x00000000	0x0000000000000000
\$f29	0x00000000	
\$f30	0x00000000	0x0000000000000000
\$f31	0x00000000	

Condition Flags			
<input type="checkbox"/> 0	<input type="checkbox"/> 1	<input type="checkbox"/> 2	<input type="checkbox"/> 3
<input type="checkbox"/> 4	<input type="checkbox"/> 5	<input type="checkbox"/> 6	<input type="checkbox"/> 7

(10001<sub>2</sub>). Поле «cop» равно 16 (10000<sub>2</sub>) для команд одинарной точности и 17 (10001<sub>2</sub>) для команд двойной точности. Аналогично командам типа R, у команд типа F есть два операнда-источника (fs и ft) и один операнд-назначение (fd).

Команды одинарной и двойной точности различаются суффиксом в мнемонике (.s и .d соответственно). Команды с плавающей точкой включают:

- сложение (add.s, add.d),
- вычитание (sub.s, sub.d),
- умножение (mul.s, mul.d),
- деление (div.s, div.d),
- изменение знака (neg.s, neg.d),
- вычисление модуля (abs.s, abs.d).

В тех случаях, когда ветвление в программе зависит от условия, вычисляемого с использованием ЧПЗ, оно выполняется в два этапа. Сначала команда сравнения устанавливает или сбрасывает специальный флаг условия `frcond` (от англ. floating point condition flag). После этого команда ветвления проверяет этот флаг и в зависимости от его состояния осуществляет переход.

Команды сравнения включают команды:

- проверки на равенство (c.seq.s/c.seq.d),
- проверки на то, что один операнд меньше другого (c.lt.s/c.lt.d)
- проверки на то, что один операнд меньше или равен другому (c.le.s/c.le.d).

Команды ветвления `bc1f` и `bc1t` осуществляют переход, если флаг `frcond` имеет значение ЛОЖЬ или ИСТИНА соответственно.



В симуляторе MARS моделируется MIPS-процессор, где таких флагов условий м.б. 8 штук. По умолчанию устанавливается и проверяется 0-й флаг.

```
c.eq.s $f1,$f2
```

```
bc1t label
```

Однако при программировании можно специальной константой указать на то, какой именно флаг использовать (в примере ниже используется 1-й флаг)

```
c.eq.s 1,$f1,$f2
```

```
bc1t 1,label
```

Регистры с плавающей точкой загружаются из памяти и записываются в память при помощи команд `lwc1` и `swc1` соответственно. Эти команды перемещают по 32 бита, так что для работы с числами двойной точности необходимо использовать по две такие команды.

Нет никаких специальных констант и функций, точность исходных данных возлагается на пользователя. Отсутствуют инструкции для вычисления алгебраических функций, расчеты при необходимости пользователь должен будет программировать самостоятельно (например, с помощью числовых рядов).

Рассмотрим, программу для MIPS, которая по примеру со стр.11 также вычисляет сумму двух дробных чисел и заносит результат в память. Дополнительно переменные выводятся в окно сообщений MARS.

```
.data
aa:   .float  22.1
bb:   .float  33.7
Sum:  .float  0.0
head_A: .asciiz "\n A=\0"
head_B: .asciiz "\n B=\0"
head_Sum: .asciiz "\n Sum A+B=\0"
```

```
.text
la $a0, head_A    # load address of print heading
li $v0, 4         # specify Print String service
syscall          # print head_A
```

```

lwc1 $f1, aa    # load number A from memory in register $f1
lwc1 $f12, aa   # load number A from memory in register $f12 for printing
li $v0, 2       # specify Print Float service
syscall        # print float A (from $f12)

la $a0, head_B  # load address of print heading
li $v0, 4       # specify Print String service
syscall        # print head_B

lwc1 $f2, bb    # load number B from memory in register $f2
lwc1 $f12, bb   # load number B from memory in register $f12 for printing
li $v0, 2       # specify Print Float service
syscall        # print float B (from $f12)

add.s $f12, $f2, $f1, # calculate $f3=$f1+$f2
swc1 $f12, Sum

la $a0, head_Sum # load address of head_Sum
li $v0, 4        # specify Print String service
syscall        # print head_Sum
li $v0, 2        # specify Print Float service
syscall        # print Sum number

```

Результатом работы программы будет

The screenshot shows a debugger window with a 'Data Segment' table and a 'Mars Messages' window. The Data Segment table has columns for Address, Value (+0), Value (+4), Value (+8), Value (+c), Value (+10), and Value (+14). The Mars Messages window shows the output of the program: A=22.1, B=33.7, Sum A+B=55.800003, and a message indicating the program is finished running.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10010000	0x41b0cccd	0x4206cccd	0x425f3334	0x4120200a	0x0a00003d	0x3d422b41
0x10010020	0x3d422b41	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Mars Messages: Run I/O

Clear

```

A=22.1
B=33.7
Sum A+B=55.800003
-- program is finished running (dropped off bottom) --

```

## 2. ЗАДАНИЕ

1) Запишите число  $A = \pm a_1 a_2 a_3 a_4, a_5 a_6$  в десятичной системе счисления, представляющее дату вашего рождения, где

$a_1 a_2$  – ГОД

$a_3 a_4$  – МЕСЯЦ

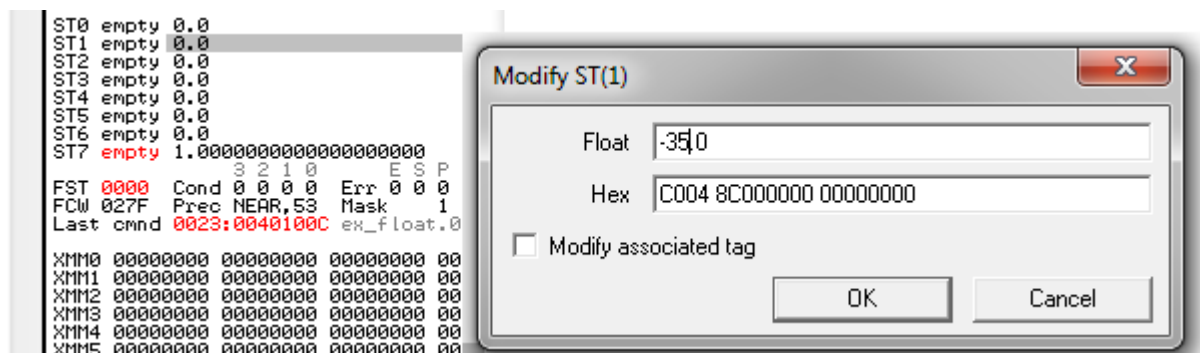
$a_5 a_6$  – ЧИСЛО,

знак числа выберите по правилу: минус, если  $a_4$  – чётное, иначе – плюс.

2) Представьте это число  $A$  в формате IEEE754-2008 с одинарной, двойной и расширенной точностью,

3) Переведите получившееся многобайтовое число из двоичной в шестнадцатеричную систему счисления;

4) Проверьте ваши вычисления. Для этого можно создать программу на любом языке программирования, которая выведет на печать последовательно все байты числа для каждого формата (float, double, long double). Либо можно воспользоваться сервисом перевода на сайте «Википедия» в разделе «IEEE 754-2008/ IEEE754 онлайн двоично-десятичный преобразователь» или аналогичными интернет-сервисами. Либо в отладчике OllyDbg кликнув два раза мышкой по значению любого ST регистра в выпадающем окне задать исходное десятичное представление числа и увидеть автоматически формирующееся шестнадцатеричное представление (не перевёрнутое).



5) Напишите отчёт о проделанной работе, куда включите пункты 1,2,3 и распечатку с результатами программы из пункта 4.

6) Создайте свою программу **prakt\_float.asm** согласно варианту в редакторе Quick Editor пакета MASM, которая выполняет заданные действия с

действительными величинами. Помните, что при копировании текста из других редакторов некоторые особенности форматирования могут рассматриваться как синтаксические ошибки в ассемблерной программе (язык ввода, точки, запятые, минус...).

- 7) Откомпилируйте программу, исправьте ошибки, запустите на исполнение в OllyDbg, проследите изменения в регистрах, памяти.
- 8) Найдите в памяти все использованные в программе переменные и определите знак в их шестнадцатеричном представлении.
- 9) Рассчитайте погрешность результата компьютерных вычислений
- 10) Ответьте на вопросы для самопроверки.

### 3. ВАРИАНТЫ ЗАДАНИЙ

*Работа может быть выполнена группой из двух студентов.*

***Используйте вычисленное вами число А, при необходимости В возьмите любое. Для неподдерживаемых процессором MIPS констант вычислите их программно или задайте их вручную (сравните точность с аналогичными данными для Intel). Как и в предыдущих случаях надо уметь объяснить, как выполнить любой из вариантов задания.***

№ вар	Задание (целочисленные операции)
1.	Выделить из переменной А в формате ЧПЗ порядок и мантиссу. Воспользуйтесь командой fxtcat (Intel). Результат М и Р сохранить в памяти как новые переменные.
2.	Вычислить выражение $S=(A:4+\log_2 e) \times B$ . Воспользуйтесь командой fldl2e (Intel). Результат S сохранить в памяти как новую переменную.
3.	Сравнить два вещественных числа А и В (воспользуйтесь командами fcomi, fucomi, fucomip – Intel или c.seq./c.lt.s/c.le.s/c.bclf/bc1t – для MIPS) и вычислить выражение $S=\max(A,B)+\frac{1}{4} \times \min(A,B)$ . Результат S сохранить в памяти как новую переменную.

4.	Вычислить выражение $S= A+B :2$ . Воспользуйтесь командой <code>fabs</code> (Intel) или <code>abs.s/abs.d</code> для MIPS. Результат $S$ сохранить в памяти как новую переменную.
5.	Вычислить площадь круга $S$ с радиусом $A$ . Воспользуйтесь командой <code>fldpi</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
6.	Вычислить выражение $S=\sqrt{A} + 1$ . Воспользуйтесь командами <code>fldl1</code> , <code>fsqrt</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
7.	Вычислить выражение $S=(2^B-1)+A \times 8$ . Воспользуйтесь командой <code>f2xm1</code> и <code>fld1</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
8.	Вычислить площадь $S$ треугольника по двум сторонам ( $A$ , $B$ ) и углу между ними. Воспользуйтесь командой <code>fsin</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
9.	Вычислить площадь прямоугольного треугольника по известным катету и прилежащему углу. Воспользуйтесь командой <code>fcos</code> или <code>fsin</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
10.	Вычислить выражение $S=(A \times 4 - B) : \ln 2$ . Воспользуйтесь командой <code>fldln2</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
11.	Вычислить выражение $S=(1+\log_{10} 2) \times A$ . Воспользуйтесь командами <code>fldlg2</code> , <code>fld1</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.
12.	Вычислить выражение $S=(1-\ln 2) \times A$ . Воспользуйтесь командами <code>fldln2</code> , <code>fld1</code> (Intel). Результат $S$ сохранить в памяти как новую переменную.

#### 4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как происходит работа с дробными числами?
2. Как хранятся дробные числа в памяти компьютера? Найдите все части числа ЧПЗ: знак, порядок, мантиссу на примере своего числа  $A$ .
3. Какие регистры и сколько используются для работы с дробями?
4. Для чего используются теги регистров сопроцессора (Intel)?

5. Какие стандартные константы и функции используются сопроцессором при работе с ЧПЗ (Intel)?
6. Как организован набор регистров сопроцессора (Intel)?
7. Какой флаг можно использовать при работе с ЧПЗ в MIPS?
8. Сравните два ЧПЗ (по шестнадцатеричным HEX-кодам их байтов в памяти): больше, меньше, равны, положительные, отрицательные.
9. Объясните на примере вашей программы изменения в памяти и регистрах процессора и сопроцессора.

## **5. СПИСОК ЛИТЕРАТУРЫ**

1. Intel 64 and IA32-Architectures Software Developers Manual (с сайта intel.com)
2. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. – СПб.: Издательство «Питер», 2000.
3. Юров В. Ассемблер. – СПб.: Издательство «Питер», 2002.
4. Д. М. Харрис и С. Л. Харрис. Цифровая схемотехника и архитектура компьютера. Morgan Kaufman © English Edition, 2013г.
5. <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>.
6. Сайт «Учебный курс: Методы | Регистры MIPS»
7. Скляр И. Изучаем Ассемблер за 7 дней. 2010 - стр. 156-158

## ГЛАВА 5. СПЕЦИАЛЬНЫЕ ТИПЫ ДАННЫХ: ВЕКТОРНЫЕ ОПЕРАЦИИ, ДВОИЧНО-ДЕСЯТИЧНАЯ АРИФМЕТИКА.

### Практическая работа №5

#### 1. ТЕОРИЯ

##### 1.1. Десятичная арифметика

*Десятичные числа* (или правильнее двоично-десятичные) – специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD – Binary-Coded Decimal).

Двоично-десятичный код, BCD, 8421-BCD – форма записи целых чисел, когда каждый десятичный разряд числа записывается в виде его четырёхбитного двоичного кода.

Например, десятичное число  $311_{10}$  будет записано в двоичной системе счисления в двоичном коде как  $1\ 0011\ 0111_2$ , а в двоично-десятичном коде как  $0011\ 0001\ 0001_{BCD}$ .

#### Преимущества

- Значения стандартных величин (особенно в формате слова и двойного слова) имеют ограниченный диапазон, тогда как запрограммировать BCD-число любой размерности не составит труда.
- Для дробных чисел (как с фиксированной, так и с плавающей запятой) при переводе в человеко-читаемый десятичный формат и наоборот не теряется точность. При операциях над двоичными числами всегда возникают ошибки округления (для целых – при выходе за пределы разрядной сетки, для дробных – 1) при выходе за пределы разрядной сетки, 2) при переводе десятичной дроби в двоичную при вводе и обратно при выводе). Тогда как несложно запрограммировать выполнение любых операций над BCD-числами без перевода в машинную двоичную систему счисления.

- Представление большого объема результатов вычислений (десятичных чисел) при выводе на индикацию в символьном виде (ASCII-коде) – довольно сложно. Перевод чисел из двоичного кода в ASCII-код требует определенных вычислительных затрат<sup>5</sup>. Для BCD-чисел упрощён вывод на индикацию – вместо последовательного деления на 10 требуется просто вывести каждую десятичную цифру (полубайт) с небольшим изменением<sup>6</sup>. Аналогично, проще ввод данных с цифровой клавиатуры.
- Упрощены операции умножения и деления на 10, а также округления.

### Недостатки

- Требуется больше памяти.
- Усложнены арифметические операции, так как в BCD-числах (например, код 8421) используются только 10 возможных комбинаций 4-х битового поля вместо 16, существуют запрещённые комбинации битов: 1010( $10_{10}$ ), 1011( $11_{10}$ ), 1100( $12_{10}$ ), 1101( $13_{10}$ ), 1110( $14_{10}$ ) и 1111( $15_{10}$ ), которые не соответствуют никакой десятичной цифре, но могут появиться в ходе вычислений, а значит требуют использования дополнительных корректирующих команд.

По этим причинам двоично-десятичный формат применяется в бизнес-приложениях (где числа должны быть большими и точными) и калькуляторах (которые в простейших арифметических операциях должны выводить в точности такой же результат, какой подсчитает человек на бумаге).

---

<sup>5</sup> – При переводе переменной (обычного двоичного числа) в строку символов для последующего отображения на экране в виде десятичного числа требуется последовательно делить переменную на 10 и формировать из остатка очередную десятичную цифру. Для действительных значений операция вывода усложняется многократно.

<sup>6</sup> – Если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры и на соответствующий ей ASCII-код, то видно, что они отличаются на величину  $30h$ . Таким образом, преобразование в символьный вид (и обратно) получается прибавлением (или вычитанием) величины  $30h$ .



Микропроцессор хранит BCD-числа в двух форматах (рис. 1):

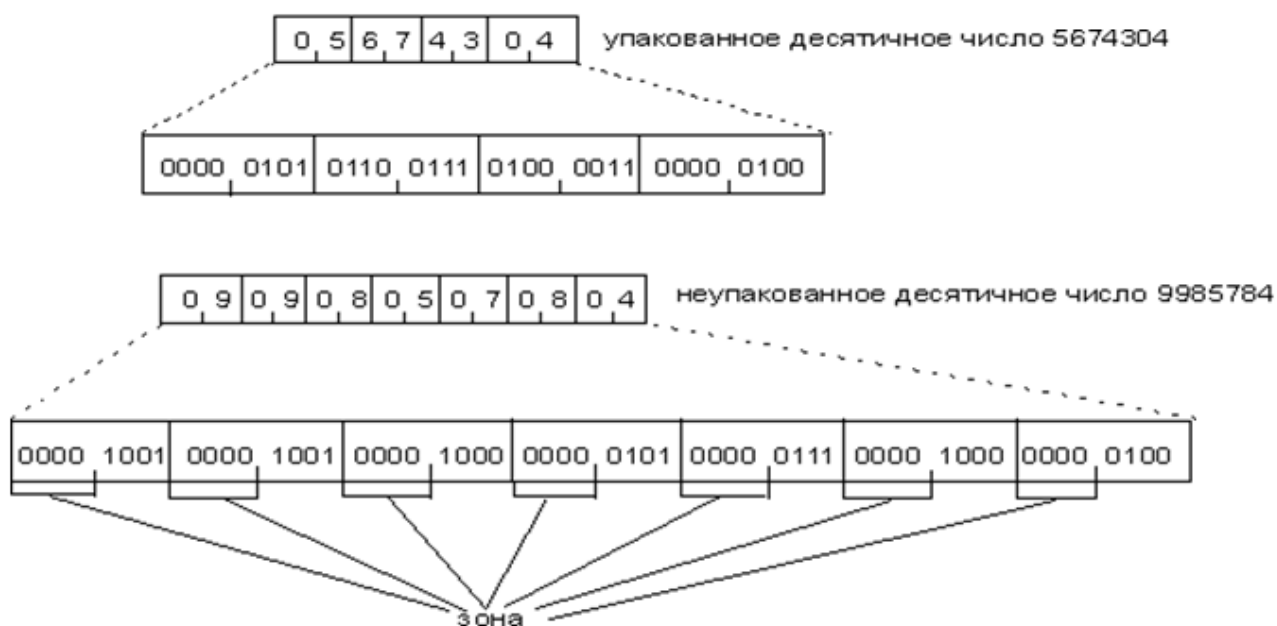


Рис. 1. Представление BCD-чисел

- **упакованном формате** – в этом формате каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 бита. При этом код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;
- **неупакованном формате** – в этом формате каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая зона. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

## 1.2. Программирование на Ассемблере Intel

Как описать двоично-десятичные числа в программе? Правильнее использовать только две директивы описания и инициализации данных – db (1байт) и dt (10байт). Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип “младший байт по младшему адресу”, что очень удобно для их

обработки. Но допустимы и другие формы описания – это дело вкуса и личных пристрастий программиста.

Например, приведенная в секции данных последовательность описаний BCD-чисел будет выглядеть в памяти так.

Address	Hex dump	ASCI
00403000	45 56 87 09 00 00 00 00 00 00 02 03 04 06 08 02	EUS.
00403010	05 00 06 00 00 00 00 00 00 00 00 00 00 00 00 00	⚡.⚡.
00403020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

оба числа перевернутые

Расположение BCD-чисел в памяти

.data

a\_pac dt **9875645**h ;упакованное BCD-число 00000000000009875645  
a\_unpac db **2,3,4,6,8,2** ; неупакованное BCD-число 286432

← младший байт                      ← старший байт

### 1.3. Арифметические операции над BCD-числами

В Ассемблере Intel-32 не существует специальных команд для операций над BCD-числами, они выполняются теми же командами, что и операции с двоичными целыми числами. Однако используются специальные команды для коррекции полученного этими командами результата ( $\pm 0110$  в неправильной тетраде). С помощью одной стандартной команды и одной специальной корректирующей команды можно выполнить только шесть основных арифметических операций над одноразрядными BCD-числами:

- сложение неупакованных BCD-чисел;
- вычитание неупакованных BCD-чисел;
- умножение неупакованных BCD-чисел;
- деление неупакованных BCD-чисел;
- сложение упакованных BCD-чисел;
- вычитание упакованных BCD-чисел.

При сложении и вычитании чисел формата 8421-BCD действуют следующие правила:

1. когда происходит перенос бита в старшую тетраду, необходимо к тетраде, от которой произошёл перенос, добавить корректирующее значение 0110 ( $= 6_{10} = 16_{10} - 10_{10}$ : разница количеств комбинаций тетрады и используемых значений).
2. При сложении двоично-десятичных чисел каждый раз, когда встречается недопустимая комбинация, необходимо добавить корректирующее значение 0110 с разрешением переноса в старшие полубайты.
3. При вычитании двоично-десятичных чисел, для каждого полубайта, получившего заём из старшего полубайта, необходимо провести коррекцию, отняв значение 0110.

Именно эти операции с соответствующими проверками выполняют специализированные команды, применять которые нужно сразу же после выполнения операции над BCD-числом.

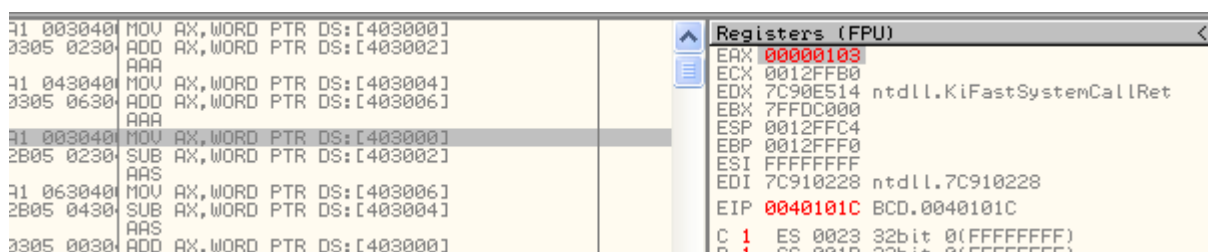
При программировании операций над упакованными BCD-числами программист должен САМ ОСУЩЕСТВЛЯТЬ КОНТРОЛЬ ЗА ЗНАКОМ ЧИСЛА.

#### 1.4. Сложение неупакованных BCD-чисел

```
.data
A   DB   07H, 09h ;неупакованное BCD-число 97
B   DB   06H, 01h ;неупакованное BCD-число 16
D   DB   dup (3)
;-----
.code
...
MOV AL,A   ;перемещение числа A в регистр AL
ADD AL,B   ;сложение AL с числом B
AAA        ;коррекция результата сложения неупакованных BCD-чисел
```

**AAA (ASCII Adjust for Addition)** – коррекция результата сложения неупакованных BCD-чисел для представления в символьном виде. Эта команда не имеет операндов. Она работает неявно только с регистром AL и анализирует значение его младшей тетрады:

- если это значение меньше 9, то флаг CF сбрасывается в 0 и осуществляется переход к следующей команде;
- если это значение больше 9, то выполняются следующие действия:
  - к содержимому младшей тетрады AL (но не к содержимому всего регистра!) прибавляется 6, тем самым значение десятичного результата корректируется в правильную сторону;
  - флаг CF устанавливается в 1, тем самым фиксируется перенос в старший разряд, для того чтобы его можно было учесть в последующих действиях.



В рассмотренном примере значение суммы  $7+6=13=0000\ 1101$  находится в AL, а после команды AAA в регистре будет  $1101 + 0110 = 0011$ , то есть двоичное 0000 0011 или десятичное 3, а флаг **CF** установится в 1, то есть **перенос** запомнился в микропроцессоре. Далее программисту нужно будет использовать команду сложения ADC, которая учтет перенос из предыдущего разряда.

```
MOV AL,A[1]
ADC AL,B[1]
AAA
```

### 1.5. Вычитание неупакованных BCD-чисел

В случае, когда вычитаемое меньше уменьшаемого проблем не возникает:

```
.data
A DB 09H
```

```
B DB 06H
```

```
;-----
```

```
.code
```

```
MOV AL,A
```

```
SUB AL,B ;9-3=6
```

```
AAS ; коррекция результата вычитания неупакованных BCD-чисел
```

В случае, когда уменьшаемое меньше вычитаемого, происходит заём из старшего десятичного разряда

```
.data
```

```
A DB 03H
```

```
B DB 06H
```

```
.code
```

```
MOV AL,A
```

```
SUB AL,B ;3<6 вычесть нельзя, а с учётом заёма можно: 13-6=7
```

```
AAS ;коррекция результата вычитания неупакованных BCD-чисел
```

На самом деле происходит хитрая вещь. Сначала выполняется обычное вычитание и формируется отрицательный результат в дополнительной коде

$$3-6=[-3]_{\text{доп}}=FD.$$

Затем корректирующая команда **AAS** рассматривает результат и определяет требуется ли коррекция и указание на заём из старшего десятичного разряда (выставляя при этом флаг CF в 1), который следует в дальнейшем учесть при продолжении операции вычитания со старшими байтами, т.е. использовать команду **SBB** – вычитание с заёмом.

**AAS (ASCII Adjust for Substraction)** – коррекция результата вычитания неупакованных BCD-чисел для представления в символьном виде.

Команда **AAS** также не имеет операндов и работает с регистром **AL**, анализируя его младшую тетраду следующим образом:

- если ее значение меньше 9, то флаг CF сбрасывается в 0 и управление передается следующей команде;

- если значение тетрады в AL больше 9, то команда AAS выполняет следующие действия:

1. из содержимого младшей тетрады регистра AL (заметьте – не из содержимого всего регистра) вычитает 6;
2. обнуляет старшую тетраду регистра AL;
3. устанавливает флаг CF в 1, тем самым фиксируя воображаемый заем из старшего разряда.

В нашем примере результат  $AL=FD_{16}$  (младшая тетрада=D). Значит надо корректировать:

1. мл. тетрада AL:  $D_{16} - 6 = 7$ ;
2. старшая тетрада AL = 0;
3. флаг CF = 1 (заем).

```
MOV AX, WORD PTR DS:[403000]
SUB AX, WORD PTR DS:[403002]
```

```
Registers (CPU)
EAX 0000FD
ECX 0012F0
EDX 7C90E514 ntd
EBX 75500000
```

после команды SUB

```
SUB AX, WORD PTR DS:[403000]
AAS
```

```
Registers (CPU)
EAX 000007
ECX 0012F0
EDX 7C90E514 ntd
EBX 75500000
```

после команды AAS

Понятно, что команда AAS применяется вместе с основными командами вычитания SUB и SBB. При этом команду SUB есть смысл использовать только один раз, при вычитании самых младших цифр операндов, далее должна применяться команда SBB, которая будет учитывать возможный заем из старшего разряда.

```
MOV AL,A+1
SBB AL,B+1
AAS
```

Результат выглядит так:

Address	Hex dump
00403000	03 09 06 01 07 07 00
	<u>03 09</u> <u>06 01</u> <u>07 07</u> 00
	<b>A</b> <b>B</b> <b>D</b>

$$\begin{array}{r}
 93A \\
 - 16B \\
 \hline
 77D
 \end{array}$$

## 1.6. Умножение неупакованных BCD-чисел

Стандартных алгоритмов для выполнения этих действий над BCD-числами нет, и программист должен сам, исходя из требований к своей программе, реализовать эти операции. Реализация умножения и деления – еще более сложна. В системе команд микропроцессора присутствуют только средства для производства умножения и деления одноразрядных неупакованных BCD-чисел. Для того чтобы умножать числа произвольной размерности, нужно реализовать процесс умножения самостоятельно, взяв за основу некоторый алгоритм умножения, например, “в столбик” или ускоренного умножения/деления.

Для того чтобы перемножить два одноразрядных BCD-числа, необходимо:

- поместить один из сомножителей в регистр AL (как того требует команда MUL);
- поместить второй операнд в регистр или память, отведя байт;
- перемножить сомножители командой MUL (результат, как и положено, будет в AX);
- результат, конечно, получится в двоичном коде, поэтому его нужно скорректировать.

Для коррекции результата после умножения применяется специальная команда

**AAM (ASCII Adjust for Multiplication)** – коррекция результата умножения для представления в символьном виде. Она не имеет операндов и работает с регистром AX следующим образом:

- делит AL на 10;
- результат деления записывается так: частное в AH, остаток в AL.

В результате после выполнения команды AAM в регистрах AL и AH находятся правильные двоично-десятичные цифры произведения двух цифр.

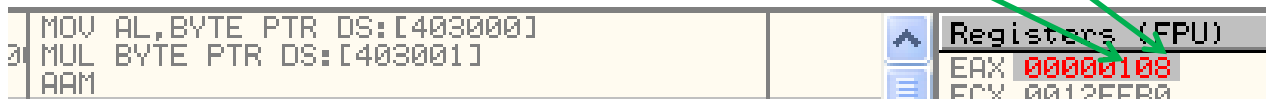
```
.data
A   DB  03H
B   DB  06H
;-----
.code
```

...

MOV AL,A

MUL B

AAM  $3 \times 6 = 18_{10} = 12_{16}$   $18:10 = 1(8 \text{ в ост}) \rightarrow AH=1, AL=8$



## 1.7. Деление упакованных BCD-чисел

Процесс выполнения операции деления двух упакованных BCD-чисел несколько отличается от других, рассмотренных ранее, операций с ними. Здесь также требуются действия по коррекции, но они должны выполняться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре AX нужно получить две упакованные BCD-цифры делимого. Это делает программист удобным для него способом. Далее нужно выдать команду AAD:

**AAD (ASCII Adjust for Division)** – коррекция деления для представления в двоичном виде. Команда не имеет операндов и преобразует двузначное упакованное BCD-число в регистре AX в двоичное число. Это двоичное число впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда AAD помещает полученное двоичное число в регистр AL. Делимое, естественно, будет двоичным числом из диапазона 0...99.

Алгоритм, по которому команда AAD осуществляет это преобразование, состоит в следующем:

- умножить старшую цифру исходного BCD-числа в AX (содержимое AH) на 10;
- выполнить сложение AH + AL, результат которого (двоичное число) занести в AL;
- обнулить содержимое AH.



Далее программисту нужно выдать обычную команду деления DIV для выполнения деления содержимого AX на одну BCD-цифру, находящуюся в байтовом регистре или байтовой ячейке памяти.

.data

B DW 0306H ;неупакованное двухбайтовое BCD-число 36

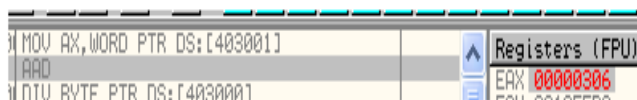
A DB 3

...

MOV AX,B

AAD ;коррекция перед делением

DIV A ;в AL BCD-частное, в AH BCD-остаток



Аналогично AAM, команде AAD можно найти и другое применение – использовать ее для перевода неупакованных BCD-чисел из диапазона 0...99 в их двоичный эквивалент.

Для деления чисел большей разрядности, так же как и в случае умножения, нужно реализовывать свой алгоритм, например “в столбик”, либо найти более оптимальный путь.

## 1.8. Арифметические действия над упакованными BCD-числами

Как уже отмечалось выше, упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнительно преобразовывать либо в неупакованный формат, либо в двоичное представление.

## 1.9. Сложение упакованных BCD-чисел

Как и для неупакованных BCD-чисел, для упакованных BCD-чисел существует потребность как-то корректировать результаты арифметических операций. Микропроцессор предоставляет для этого команду DAA:

**daa (Decimal Adjust for Addition)** – коррекция результата сложения упакованных BCD-чисел для представления в десятичном виде. Команда DAA преобразует содержимое регистра AL в две упакованные десятичные. 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 мл. тетрады.

Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается в флаге CF, тем самым учитывается перенос в старший разряд.

$$\begin{array}{r}
 +67 \quad = 0110\ 0111 \\
 \underline{75} \quad = 0111\ 0101 \\
 \hline
 \quad = \underline{1101\ 1100} - \text{требуется коррекция} \\
 \quad + 0110\ 0110 \\
 \text{CF}=1 \leftarrow 0100\ 0010 \\
 \quad 1 \quad 4 \quad 2
 \end{array}$$

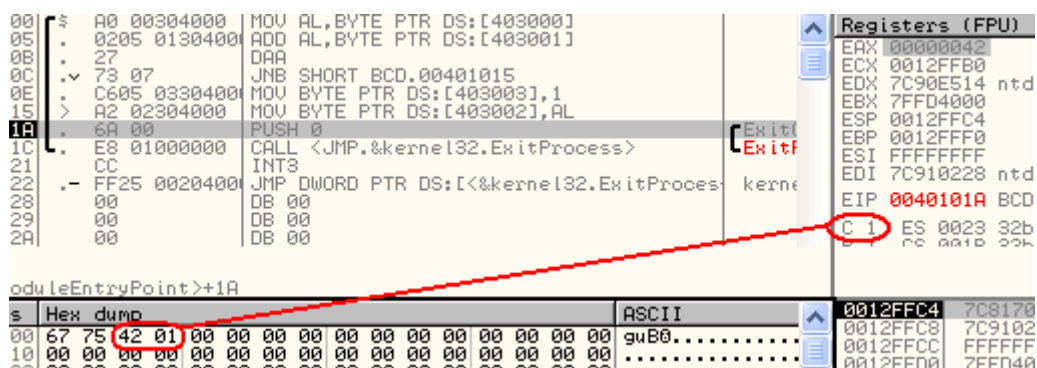
Проиллюстрируем сказанное на примере сложения двух двузначных BCD-чисел в упакованном формате: 67+75=142.

.data

```

B    DB    67H    ;упакованное число 67h
A    DB    75H    ;упакованное число 75
SUM  DB    2 DUP (0)
;-----
.code
MOV  AL,B
ADD  AL,A
DAA
JNC  MET        ;переход через команду, если был перенос (CF=1)
MOV  SUM+1,1    ;учет переноса при сложении (результат > 99)
MET: MOV  SUM,AL

```



В приведенном примере все достаточно прозрачно, единственное, на что следует обратить внимание, – это описание упакованных BCD-чисел и порядок формирования результата. Результат формируется в соответствии с основным принципом работы микропроцессоров Intel: младший байт по младшему адресу.

### 1.10. Вычитание упакованных BCD-чисел

Аналогично сложению, микропроцессор рассматривает упакованные BCD-числа как двоичные и, соответственно, выполняет вычитание BCD-чисел как обычных двоичных байтов. Само вычитание BCD-чисел осуществляется стандартной командой вычитания SUB или SBB. Коррекция результата после вычитания осуществляется командой DAS.

**DAS (Decimal Adjust for Subtraction)** – коррекция результата вычитания упакованных чисел для представления в десятичном виде. Команда DAS преобразует содержимое регистра AL в две упакованные десятичные цифры.

Команда DAS 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 заёма 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 уменьшается на 06;
- для ситуации 2 содержимое регистра AL уменьшается на 60h;
- если имеют место обе ситуации, то корректировка начинается с младшей тетрады.

заём в старшем разряде

● ● ● ● ●

```

_ 167 = 0000 0001 0110 0111
  75  =                0111 0101
-----
                        1111 0010  запрещённая комбинация
                        - 0110      коррекция
                        1001 0010
                          9   2
  
```

```

.data
b db 67h ;упакованное число
a db 75h ;упакованное число
raz db 2 dup (0)
;-----
.code
start:
mov al,b
sub al,a
das
  
```

### 1.11. Отрицательные BCD-числа

Если требуется задать отрицательное BCD-число, то надо самостоятельно задать дополнительные переменные, хранящие знаки чисел, над которыми выполняются операции и перед вычислением проверять абсолютные значения чисел. Далее, вычитать следует из большего меньшее и самостоятельно следить за знаком результата (по большему из чисел).

### 1.12. Многобайтовые BCD-числа

Для выполнения действий над большими BCD-числами следует организовать цикл по количеству байт в числах (или по самому короткому из них) и в цикле выполнять требуемую операцию многократно, учитывая перенос/заём, сохраняя результат в заранее выделенное место в памяти. Ниже приведён пример сложения двух неупакованных трёхбайтовых BCD-чисел.

```
.data
A   DB  07H, 09h, 04H   ;неупакованное BCD-число 497
B   DB  06H, 01h, 05H   ;неупакованное BCD-число 516
D   DB  4 dup (0)       ;неупакованный результат, размером на 1 байт больше
;-----
.code
start:
    MOV ECX,3           ; количество байт в складываемых числах
    XOR ESI,ESI        ; № складываемого байта числа
    CLC                ; очистка флага CF (CF:=0)
FOR1: MOV AL,A[ESI]    ;перемещение очередного байта числа A в регистр AL
    ADC AL,B[ESI]      ;сложение AL с очередным байтом числа B
    AAA                ;коррекция результата сложения неупакованных BCD-чисел
    MOV D[ESI],AL      ;перемещение очередного байта результата в переменную D
    INC ESI
    LOOP FOR1
    JNC END1
    MOV D[ESI],01H     ;сохранение последнего переноса в результат
END1:
    invoke ExitProcess, NULL
end start              ; Конец программы
```

На рисунке ниже показано содержимое памяти:

Первые 3 байта (выделено синим) – число A, следующие 3 байта (выделено зеленым) – число B, следующие 4 байта – результат D (выделено красным).

Address	Hex dump
00403000	07 09 04 06 01 05 03 01 00 01 00 00

	4	9	7	A
+	5	1	6	B
	1	0	1	3
				D

### 1.13. Вывод BCD-числа

Для вывода BCD-числа (кроме форматированного вывода) можно воспользоваться простым способом преобразования одной BCD-цифры в один символ. Помня о том, что десятичная цифра кодируется в виде ASCII-символа согласно следующему правилу:

цифра	0	1	2	3	4	5	6	7	8	9
16-ричный ASCII-код	30	31	32	33	34	35	36	37	38	39

можно легко преобразовать BCD-цифру из ячейки памяти/регистра в символ, соответствующий этой цифре.

Для упакованных BCD-чисел 1 байт цифры преобразуется в 1 байт символа. Для упакованных BCD-чисел всё немного сложнее. В 1 байте содержатся две BCD-цифры. Значит сначала надо преобразовать 2 упакованные BCD-цифры в 2 упакованные BCD-цифры (преобразовать 1 байт в 2 байта), а затем, как и в первом случае, преобразовать BCD-цифру из однобайтовой ячейки памяти/регистра в символ, соответствующий этой цифре. Далее нужно заменить символы заранее заготовленной строки на коды символов, соответствующих цифрам BCD-числа. Например, рассмотрим такую строку

```
str1 db "число: XXXXXXXXX", 0
```

№ символа	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Строка символов	ч	и	с	л	о	:		X	X	X	X	X	X	X	X	X

```

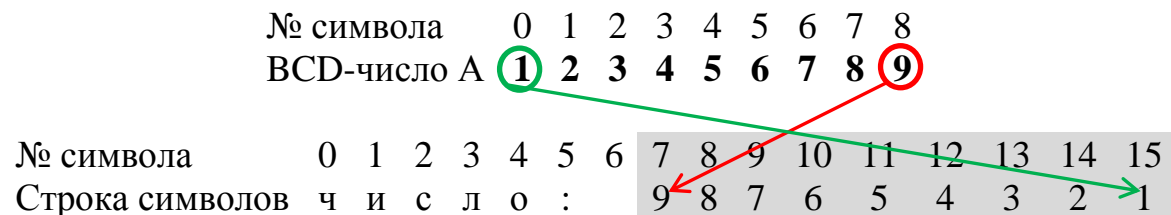
data
A db 1,2,3,4,5,6,7,8,9 ;9-байтовое BCD-число, соответствующее десятичному 987654321.
zagolovok db "Результат: ", 0
str1 db "число: XXXXXXXXXX", 0 ;строка из 16 символов
;(позднее символы XX...X будут заменены на символы, соответствующие цифрам BCD-числа)

.code
start: xor esi,esi
      mov ecx,9
m2:   mov al,A[esi] ;в цикле занесение очередной BCD-цифры в регистр AL
      or al,00110000b ;побитовое сложение с маской 30h для принудительной установки старшей тетрады в
;состояние 0011, т.е. для преобразования десятичной цифры в код символа (1->код 31, 2->код 32, и т.д.)
      mov str1[ecx+6],al ;занесение в строку str1 очередного кода цифры начиная с элемента №15 до элемента №7
;(место, с которого будут размещаться символы BCD-числа)

      inc esi
      loop m2
      invoke MessageBox, NULL, ADDR str1, ADDR zagolovok, MB_OK
stop: invoke ExitProcess, NULL

```

Согласно данному фрагменту произойдет следующее. Элементы массива заменят символы XX...X строки str1 в обратном порядке, т.к. в памяти число храниться с младшей цифры, а нам привычнее читать его со старшей цифры.



В результате вычислений получим:

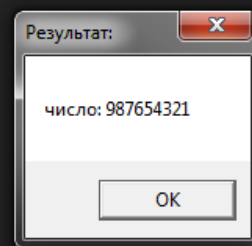
```

.data
A db 1,2,3,4,5,6,7,8,9
zagolovok db "Результат: ", 0
str1 db "число: XXXXXXXXX", 0

.code
start:      xor esi,esi
mov ecx,9
m2:        mov al,A[esi]
or al,00110000b
mov str1[ecx*6],al
inc esi
loop m2
invoke MessageBox, NULL, ADDR str1, ADDR zagolovok, MB_OK

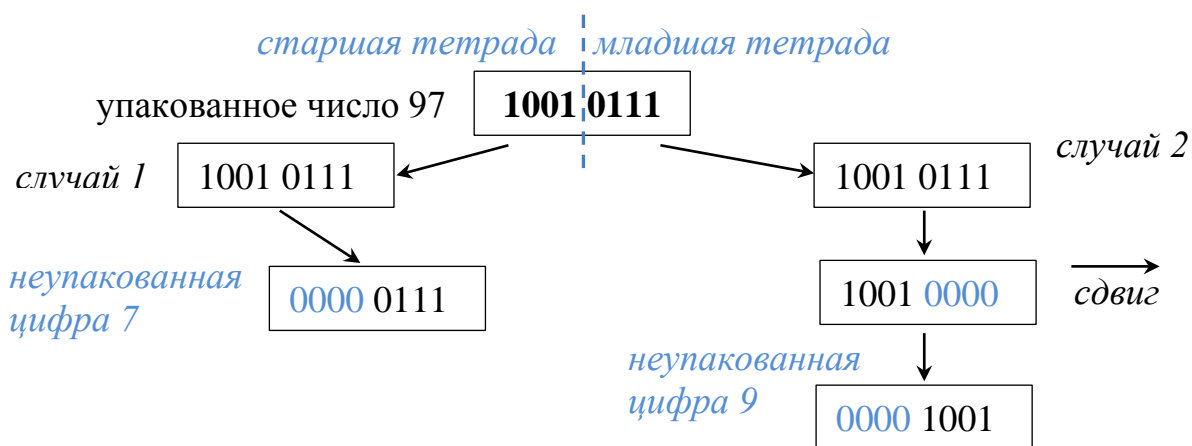
stop:      invoke ExitProcess, NULL
end start

```



В случае упакованных BCD-чисел 1 байт содержит 2 BCD-цифры. Для их разделения скопируем этот байт в два регистра и в первом обнулیم старшую тетраду, а во втором – младшую.

В первом случае мы уже получили готовую неупакованную BCD-цифру. Во втором случае эту цифру нужно сдвинуть в младшую тетраду – на 4 разряда вправо (см. рис ниже).



А дальше выполнить преобразование цифр в символ, как в и случае неупакованного BCD-числа.

```

mov al,A[esi]      ; младшая цифра
mov bl,al          ; старшая цифра
and al,00001111b  ; обнуление старшей тетрады
or al,00110000b   ; получение кода символа младшей цифры
and bl,11110000b  ; обнуление младшей тетрады
shr bl,4           ; сдвиг вправо
or bl,00110000b   ; получение кода символа старшей цифры
mov str1[ecx*2+5],bl ; сохр. старшей цифры
mov str1[ecx*2+6],al ; сохр. младшей цифры

```

В результате работы программы получим:



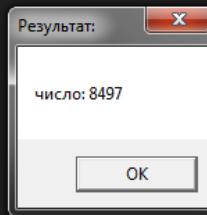
```

.data
A db 97h,84h
zagolovok db "Результат: ", 0
str1 db "число: XXXX",0

.code
start:
mov ecx,2
xor esi,esi
m2: mov al,A[esi]
mov bl,al
and al,00001111b
or al,00110000b
and bl,11110000b
shr bl,4
or bl,00110000b
mov str1[ecx*2+5],bl
mov str1[ecx*2+6],al
inc esi
loop m2

invoke MessageBox, NULL, ADDR str1, ADDR zagolovok,

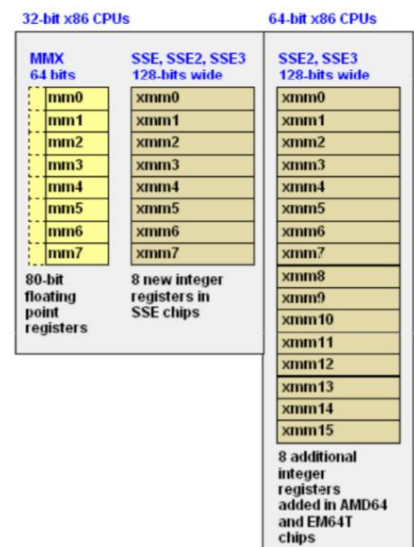
```



## 2. ВЕКТОРНЫЕ ТИПЫ ДАННЫХ

Этот тип данных предназначен для ускорения вычислений [7] и позволяет одной инструкцией закодировать параллельное выполнение одинаковых операций над набором элементов. Элементами являются упакованные в вектор данные целого типа или типа ЧПЗ.

Registers (MMX)					Registers (3DNow!)				
MM0	0105	0104	0000	0000	XMM0	00000000	00000000	00000000	00000000
MM1	0000	0000	0000	0000	XMM1	00000000	00000000	00000000	00000000
MM2	0000	0000	0000	0000	XMM2	00000000	00000000	00000000	00000000
MM3	0000	0000	0000	0000	XMM3	00000000	00000000	00000000	00000000
MM4	0000	0000	0000	0000	XMM4	00000000	00000000	00000000	00000000
MM5	0000	0000	0000	0000	XMM5	00000000	00000000	00000000	00000000
MM6	0000	0000	0000	0000	XMM6	00000000	00000000	00000000	00000000
MM7	8000	0000	0000	0000	XMM7	00000000	00000000	00000000	00000000



Векторные операции позволяют многократно увеличить производительность по сравнению со скалярными операциями (над парой чисел).

Этот тип данных появился в CISC-процессорах. Первое поколение этой технологии называлось MMX (целочисленные вектора 64 бит), далее SSE/XMM (цел. 64 бит + вектора ЧПЗ 128 бит), SSE2,3,4 (вектора 128 бит для цел. и ЧПЗ), AVX (вектора 256 бит), FMA (вектора 256/512 бит для ускоренных/слитных операций 2в1) [8].

### 2.1. Программирование под Intel

В Intel (MASM32) можно использовать два типа векторных регистров: целые MMX0÷7 и ЧПЗ XMM0÷7. Некоторые инструкции позволяют

переносить/перекодировать данные между регистрами MMX↔XMM↔GPU (EAX...). Рассмотрим простую программу для инициализации и обработки элементов вектора.

Обратите внимание на дополнительные директивы (.586 и .xmm). Векторные инструкции следует выбирать из системы команд согласно описанию исполняемой операции (проще выбирать из [5], они лучше согласуются с возможностями MASM32). Но не помешает также дополнительно просмотреть Manual [4], чтобы лучше понять правила использования той или иной инструкции для вашей программы и выбранных начальных параметров процессора.

Opcode/ Instruction	Op/ En	64 Mode	32-bit	CPUID Feature Flag	Description
F3 OF 10 /r MOVSS <i>xmm1, xmm2/m32</i>	RM	VV		SSE	Move scalar single-precision floating-point value from <i>xmm2/m32</i> to <i>xmm1</i> register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	RVM	VV		AVX	Merge scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS <i>xmm1, m32</i>	XM	VV	valid	AVX xmm	Load scalar single-precision floating-point value from <i>m32</i> to <i>xmm1</i> register.
F3 OF 11 /r MOVSS <i>xmm2/m32, xmm</i>	MR	VV		SSE	Move scalar single-precision floating-point value from <i>xmm1</i> register to <i>xmm2/m32</i> .
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS <i>xmm1, xmm2, xmm3</i>	MVR	VV		AVX	Move scalar single-precision floating-point value from <i>xmm2</i> and <i>xmm3</i> to <i>xmm1</i> register.

**.586** ;директива использования системы команд процессора Intel.586

**.xmm** ;директива использования технологии xmm

.model flat, stdcall

option casemap :none

include \masm32\include\windows.inc

include \masm32\include\user32.inc

include \masm32\include\kernel32.inc

include \masm32\include\debug.inc

includelib \masm32\lib\debug.lib

includelib \masm32\lib\user32.lib

includelib \masm32\lib\kernel32.lib

.data

vect1 dd 18567464.567, 2.8797060, 0.00008, 497877453.0, 559573462527.0, 69595745252.0, 79449383625.0, 89483271541.0

bcd1 db 4 dup(0)

bcd2 db 4 dup(0)

.code

start:

movaps XMM0, vect1 ; копирование в векторный регистр XMM0 содержимого памяти, размером 128 бит (16 Байт),  
; начиная с адреса vect1, данные интерпретируются как коды ЧПЗ одинарной точности (SP)

movaps XMM1, vect1+16 ; копирование в XMM1 четырёх ЧПЗ(SP) 16 Байт памяти, начиная с адреса vect1+16

addps XMM0, XMM1 ; сложение двух векторов из четырёх элементов ЧПЗ(SP) поэлементно:

; XMM0[0÷31]=XMM0[0÷31]+XMM1[0÷31] ... XMM0[96÷127]=XMM0[96÷127]+XMM1[96÷127]

movss bcd1, XMM0 ; копирование скалярного ЧПЗ(SP) = одного младшего элемента вектора из регистра XMM0

; в память по адресу bcd1, данные интерпретируются как простая последовательность байтов, которые в

; дальнейшем можно будет использовать по усмотрению программиста (например, как отдельные BCD-цифры)

invoke ExitProcess, NULL

end start ; окончание программы

Обратите внимание, в какой последовательности содержимое байтов памяти копируется в элементы вектора по инструкции *movaps XMM0, vect1*. Младший 0-й элемент вектора расположен слева, а старший 3-й справа.

```

MOVAPS XMM0,DWORD PTR DS:[403000]
SHUFPS XMM2,XMM0,93
SHUFPS XMM2,XMM0,3E
MOVAPS XMM0,DWORD PTR DS:[403000]
MOVAPS XMM1,DWORD PTR DS:[403010]
ADDPS XMM0,XMM1
MOVSS DWORD PTR DS:[403020],XMM0
MOVSS DWORD PTR DS:[403024],XMM2
PUSH 0
CALL <JMP.&kernel32.ExitProcess>
INT3
JMP DWORD PTR DS:[<&kernel32.ExitPi
DB 00
DB 00
DB 00
DB 00
DB 00
Registers (MMX)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
MM0 0000 0000 0000 0000
MM1 0000 0000 0000 0000
MM2 0000 0000 0000 0000
MM3 0000 0000 0000 0000
MM4 0000 0000 0000 0000
MM5 0000 0000 0000 0000
MM6 0000 0000 0000 0000
MM7 0000 0000 0000 0000
XMM0 33333333 22222222 11111111 00000000
XMM1 00000000 00000000 00000000 00000000
XMM2 00000000 33333333 22222222 11111111
XMM3 00000000 00000000 00000000 00000000
XMM4 00000000 00000000 00000000 00000000
Hex dump
00 00 00 00 11 11 11 11 22 22 22 22 33 33 33 33
F4 AC A6 41 69 FC 93 21 54 96 11 82 D4 03 3B 53
ASCII
0018FF88 75188543 C
0018FF8C 7FFDE000 .à
0018FF90 0018FFD4 Öÿ

```

Для копирования одного скалярного значения ЧПЗ(SP) из регистра XMM0 (например с помощью *MOVSS DWORD PTR DS:[403020],XMM0*) всегда выбирается младший нулевой элемент вектора. Обратите внимание на порядок Байтов согласно архитектуре «little endian».

```

00401000 MOVAPS XMM0,DWORD PTR DS:[403000]
00401007 SHUFPS XMM2,XMM0,93
0040100B SHUFPS XMM2,XMM0,3E
0040100F MOVAPS XMM0,DWORD PTR DS:[403000]
00401016 MOVAPS XMM1,DWORD PTR DS:[403010]
0040101D ADDPS XMM0,XMM1
00401020 MOVSS DWORD PTR DS:[403020],XMM0
00401028 MOVSS DWORD PTR DS:[403024],XMM2
00401030 PUSH 0
00401032 CALL <JMP.&kernel32.ExitProcess>
00401037 INT3
00401038 JMP DWORD PTR DS:[<&kernel32.ExitPi
Registers (MMX)
MM3 0000 0000 0000 0000
MM4 0000 0000 0000 0000
MM5 0000 0000 0000 0000
MM6 0000 0000 0000 0000
MM7 0000 0000 0000 0000
XMM0 533B03D4 22222222 2193FC69 8196AC74
XMM1 533B03D4 82119654 2193FC69 41AFACF4
XMM2 00000000 33333333 22222222 11111111
XMM3 00000000 00000000 00000000 00000000
XMM4 00000000 00000000 00000000 00000000
Address Hex dump
00403000 00 00 00 00 11 11 11 11 22 22 22 22 33 33 33 33
00403010 F4 AC A6 41 69 FC 93 21 54 96 11 82 D4 03 3B 53
00403020 74 AC 96 81 00 00 00 00 00 00 00 00 00 00 00 00
ASCII
0018FF88 75188543 C
0018FF8C 7FFDE000 .à
0018FF90 0018FFD4 Öÿ
0018FF94 7788AC69 i-

```

Однако, с помощью перераспределения элементов вектора по маске (*SHUFPS XMMi, XMMj, mask*) на место 0-го элемента приёмника (*XMMi*) можно переместить любой требуемый элемент из двух векторов: *XMMi* либо *XMMj*.

Например, ниже показан «циклический сдвиг вправо» элементов вектора

XMM0 и запись результата сдвига в регистр XMM2 за два шага.

SHUFPS XMM2, XMM0, 10010100b ; первый шаг

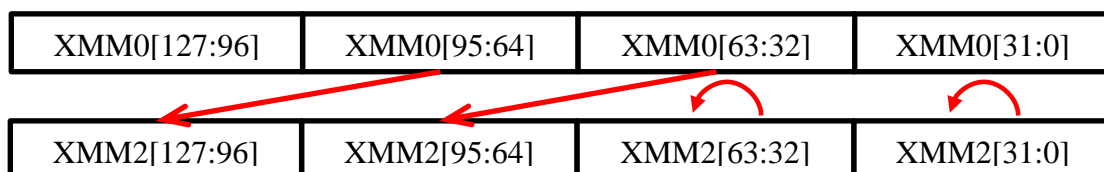
SHUFPS XMM2, XMM0, 00111110b ; второй шаг

Каждые 2 бита 8-разрядной маски определяют какие именно элементы источника и приёмника поместить в каждый элемент приёмника. Согласно двум заданным выше инструкциям выполняются следующие операции перераспределения элементов между двумя векторами.

**Состояние регистров до первого шага:**

XMM0: 3333 3333 2222 2222 1111 1111 0000 0000  
 XMM2: 0000 0000 0000 0000 0000 0000 0000 0000

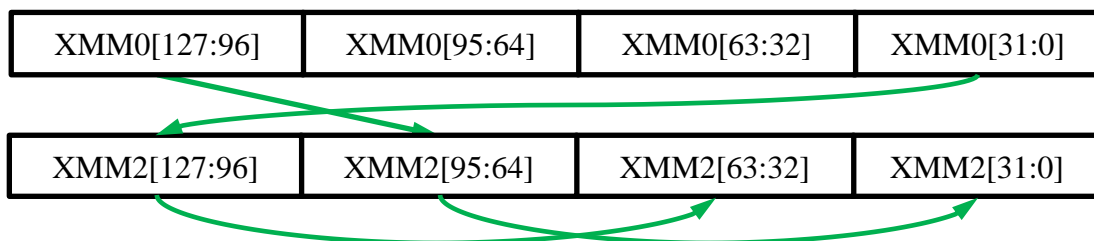
**Шаг 1**



**Промежуточное состояние регистров:**

XMM0: 3333 3333 2222 2222 1111 1111 0000 0000  
 XMM2: 2222 2222 1111 1111 0000 0000 0000 0000

**Шаг 2**



**Состояние регистров после второго шага:**

XMM0: 3333 3333 2222 2222 1111 1111 0000 0000  
 XMM2: 0000 0000 3333 3333 2222 2222 1111 1111

Результат выполнения перестановки элементов вектора можно посмотреть в отладчике во вкладке «Registers MMX».

## 2.1. Программирование под MIPS

Специально выделенных типов данных под VCD или векторные операции у MIPS нет, поэтому придётся запрограммировать эти операции и нюансы (как например, VCD-коррекция) самостоятельно.

### 3. ЗАДАНИЕ

*Работа может быть выполнена группой из двух студентов.*

Создать, отладить программу согласно варианту задания с выводом результата. Проследить в отладчике OllyDbg/MARS изменения в регистрах и ячейках памяти.

**Задача.** В памяти задан массив действительных чисел из 4-байтовых элементов (ЧПЗ SP). Загрузить последовательно расположенные элементы массива в 2 вектора. Выполнить с ними операцию согласно вашему варианту задания (табл.1). В получившемся результате выделить 0-й,  $i$ -й и  $j$ -й элементы вектора результата (табл.2) и сохранить их как три BCD-числа (тип числа  $t$ : упакованное или нет выбрать согласно табл.3). Если в записи чисел присутствуют недопустимые цифры – обнулите их. С помощью BCD-чисел вычислить выражение из табл.4 (предусмотреть возможность получения отрицательного результата). Получившийся результат вывести в окно.

Табл. 1. Варианты выбора векторной операции

№ вар	Векторная операция
1, 12	Умножить каждый элемент первый вектор на число, являющееся 0-м элементом второго вектора (SHUFPS, MULPS)
2, 13	Вычислить минимальные элементы из двух каждой пары элементов двух векторов (MINPS)
3, 14	Вычесть первый вектор из второго (SUBPS)
4, 15	Сложить первый вектор с перевернутым вторым вектором (SHUFPS, ADDPS)
5, 16	Умножить элементы первого вектора на элементы второго (MULPS)
6, 17	Разделить элементы первого вектора на элементы второго (DIVPS)
7, 18	Вычислить обратное значение каждого элемента вектора (SQRTPS)

<b>8, 19</b>	Выбрать максимальный элемент из каждой пары соответствующих элементов двух векторов (MAXPS)
<b>9, 20</b>	Вычислить квадратный корень каждого элемента второго вектора и записать их в первый вектор (SQRTPS)
<b>10, 21</b>	Выполнить логическое И (ANDPS) между разрядами элементов векторов
<b>11, 22</b>	Выполнить логическое ИЛИ (ORPS) между разрядами элементов векторов

Табл. 2. Варианты выбора номеров  $i$  и  $j$  элементов вектора в качестве BCD

№ эл-тов	$i=3$	$i=3$	$i=3$	$i=2$	$i=2$	$i=2$	$i=1$	$i=1$	$i=1$
	$j=1$	$j=2$	$j=3$	$j=1$	$j=2$	$j=3$	$j=1$	$j=2$	$j=3$
№ вар	<b>1, 10</b>	<b>2, 11</b>	<b>3, 12</b>	<b>4, 13</b>	<b>5, 14</b>	<b>6, 15</b>	<b>7, 16</b>	<b>8, 17</b>	<b>9, 18</b>

Табл. 3. Варианты выбора номеров  $i$  и  $j$  элементов вектора в качестве BCD

Тип BCD	№ варианта									
Упакованное	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
Неупакованное	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>	<b>15</b>	<b>16</b>	<b>17</b>	<b>18</b>	<b>19</b>	<b>20</b>

Табл. 4. Варианты операций с тремя BCD-числами: bcd1, bcd2, bcd3

№ варианта	Задание
<b>10.8, 15</b>	$bcd4 = +bcd1 + bcd2 - bcd3$
<b>11.9, 16</b>	$bcd4 = +bcd1 - bcd2 + bcd3$
<b>12.10, 17</b>	$bcd4 = +bcd1 - bcd2 - bcd3$
<b>13.11, 18</b>	$bcd4 = -bcd1 + bcd2 + bcd3$
<b>14.12, 19</b>	$bcd4 = -bcd1 + bcd2 - bcd3$
<b>15.13, 20</b>	$bcd4 = -bcd1 - bcd2 + bcd3$
<b>16.14, 21</b>	$bcd4 = -bcd1 - bcd2 - bcd3$

#### 4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое BCD-числа?
2. Для чего нужны BCD-числа?
3. Какие преимущества и сложности несет десятичная компьютерная арифметика по сравнению с двоичной?
4. Какой код используется для кодирования BCD-чисел?
5. Какие существуют форматы представления BCD-чисел? Приведите пример
6. Для чего требуется корректирующая BCD-команда?
7. Как происходит операция упакованного сложения и коррекция результата?
8. Как происходит операция неупакованного сложения и коррекция результата?
9. Как происходит операция упакованного вычитания и коррекция результата?
10. Как происходит операция неупакованного вычитания и коррекция результата?
11. Как происходит операция неупакованного деления и коррекция результата?
12. Как происходит операция неупакованного умножения и коррекция результата?
13. Расскажите о правилах выполнения операций над многобайтными BCD-числами?
14. Как выполнить вывод BCD-чисел?
15. Расскажите об изменениях в регистрах процессора и ячейках ОП по ходу выполнения вашей программы.
16. Что такое векторные типы данных?
17. Каково назначение векторных типов данных и векторных операций?
18. Какие бывают типы векторов в ВС?
19. Какие операции допустимы над векторами (с примерами из своей программы)?
20. Чем отличаются векторные операции от скалярных?
21. Какие существуют типы элементов в векторах?
22. Все ли операции для векторных регистров векторные? Если нет, то приведите пример.



## 5. СПИСОК ЛИТЕРАТУРЫ

1. Рикардо Нарваха. Введение в крэкинг с нуля, используя OllyDbg. Глава 1 (<http://pro.dtn.ru/cr.html>).
2. Intel 64 and IA32-Architectures Software Developers Manual (с сайта intel.com)
3. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. – СПб.: Издательство «Питер», 2000.
4. Юров В. Ассемблер. – СПб.: Издательство «Питер», 2002.
5. <https://habr.com/ru/post/440566/>
6. <http://www.mkurnosov.net/teaching/uploads/НРС/aoso-2010-lec4-sse.pdf>

# ГЛАВА 6. СТЕК. ВЫЗОВ ПРОЦЕДУР. ПЕРЕДАЧА ПАРАМЕТРОВ.

## Практическая работа №6

### 1. ТЕОРИЯ

#### 1.1. Программирование на Ассемблере Intel

В отличие от языков программирования высокого уровня в Ассемблере все подпрограммы (или процедуры) вызываются стандартным образом – с помощью команды CALL <имя> (имя=адрес). Как видим, эта команда не содержит никаких параметров. Все параметры должны быть либо глобальными переменными, либо помещаются в стек, который, как и глобальные переменные, доступен всем процедурам. Все регистры процессора – глобальные переменные.

```
~ ~ ~ ~ ~ ~ ~ ~
start:                ; Метка входа в программу
~ ~ ~ ~ ~ ~ ~ ~
PUSH параметр1       ;
...                  ; } сохранение параметров процедуры в стеке
PUSH параметрN       ; }
CALL SUMMA           ; команда вызова процедуры
~ ~ ~ ~ ~ ~ ~ ~

SUMMA proc           ; начало процедуры
POP ADR_RET          ; извлечение из вершины стека адреса возврата
POP параметрN        ;
...                  ; } восстановление параметров процедуры из стека
POP параметр1        ; } в обратном порядке
~ ~ ~ ~ ~ ~ ~ ~     ; } тело процедуры
PUSH ADR_RET         ; запись в вершину стека адреса возврата
RET                  ; команда возврата в основную (вызывающую) программу
SUMMA endp          ; конец процедуры
~ ~ ~ ~ ~ ~ ~ ~

end start            ; Конец программы
```

Сама команда CALL выполняет следующее:

- запоминает в стеке адрес возврата в текущую программу (EIP)
- изменяет регистр-указатель стека  $ESP=ESP-4$  (запись и чтение из стека происходит по 4 байта)
- формирует новый EIP адрес следующей команды, равный адресу вызываемой процедуры (метки из CALL)

По возвращении из подпрограммы прежнее значение глобальных переменных само не восстановиться. Если этого требует задача, программист должен об этом побеспокоиться сам перед вызовом процедуры, способной изменить значения переменных. В следующих двух фрагментах показано, как выглядят вызовы функции вывода сообщения в окно и завершения программы (из практической работы №1-3).

**Фрагмент 1) – вызов с помощью директивы invoke,**

```
.data
str1    db  "Это ваша первая программа для Win32",0
str2    db  "Assembler для Windows – это сказка!",0
        ;директива объявления секции кода
start:
~~~~~
invoke MessageBox, NULL, addr str2, addr str1, MB_OK
~~~~~
invoke ExitProcess, NULL
end start ;окончание программы
```

**Фрагмент 2) – вызов с помощью команды CALL.**

```
.data
str1    db  "It's the first your program for Win32",0
str2    db  "Assembler language for Windows is a fable!",0 .code

start:
~~~~~
```

```

pusha          ;сохранение значений всех регистров перед вызовом функции
push 0         ;
push offset str1 } ; передача через стек в требуемом порядке
push offset str2 } ; параметров функции MessageBox вывода окна
push 0         ;

```

```

call MessageBox ;вызов функции вывода окна

```

```

popa          ;восстановление значений всех регистров
              ;после возврата в исходную программу

```

~~~~~

```

push 0        ; передача через стек параметра функции ExitProcess
call ExitProcess ;вызов функции завершения программы

```

end start

Рассмотрим подробнее, как изменяется содержимое стека при вызове процедуры. На рисунке ниже зеленой стрелкой (и заливкой чёрным цветом) показано исходное состояние стека (адрес вершины стека ESP=0018FF8C).

Для примера регистры условно загружены разными значениями. Команда PUSHA сохраняет все 8 регистров POH (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI) в стек. Записи помещаются всё время в вершину стека, который растёт в сторону младших адресов.

The screenshot shows a debugger window with the following assembly code and registers:

| Address  | Hex dump  | ASCII                            |
|----------|-----------|----------------------------------|
| 00401000 | 60        | PUSHAD                           |
| 00401001 | ? 6A 00   | PUSH 0                           |
| 00401003 | ? 68 0A30 | PUSH OFFSET 0040300A             |
| 00401008 | ? 68 2E30 | PUSH OFFSET 0040302E             |
| 0040100D | ? 6A 00   | PUSH 0                           |
| 0040100F | ? E8 0800 | CALL <JMP.&user32.MessageBox>    |
| 00401014 | ? 61      | POPAD                            |
| 00401015 | ? 6A 00   | PUSH 0                           |
| 00401017 | ? E8 0600 | CALL <JMP.&kernel32.ExitProcess> |
| 0040101C | ? FF25 08 | JMP DWORD PTR DS:[<&user32.M...  |
| 00401022 | ? FF25 00 | JMP DWORD PTR DS:[<&kernel32...  |
| 00401028 | ? 0000    | ADD BYTE PTR DS:[EAX], AL        |

| Register | Value           |
|----------|-----------------|
| EAX      | 00000000        |
| ECX      | CCCCCCCC        |
| EDX      | DDDDDDDD        |
| EBX      | BBBBBBBB        |
| ESP      | 0018FF8C        |
| EBP      | 0018FF94        |
| ESI      | 11111111        |
| EDI      | FFFFFFFF        |
| EIP      | 00401000 stec.< |

| Address  | Hex dump | ASCII |
|----------|----------|-------|
| 0018FF8C | 7752338A |       |
| 0018FF90 | 7EFDE000 |       |
| 0018FF94 | 0018FFD4 |       |
| 0018FF98 | 77DD9902 |       |
| 0018FF9C | 7EFDE000 |       |
| 0018FFA0 | 76D6E6ED |       |
| 0018FFA4 | 00000000 |       |
| 0018FFA8 | 00000000 |       |

После операции сохранения регистров в стеке должно прибавиться 8 записей по 4 байта (т.к. моделируемый процессор 32 разрядный) и адрес вершины стека уменьшиться на  $8 \times 4 = 32_{10} = 20_{16}$ .  $ESP = ESP - 20_{16} = 0018FF8C - 20 = 0018FF6C$ .

**БУДУЩИЙ АДРЕС ВОЗВРАТА**

|     |
|-----|
| EDI |
| ESI |
| EBP |
| ESP |
| EBX |
| EDX |
| ECX |
| EAX |

Далее формируем в стеке список передаваемых параметров: ноль, адреса двух строк, ноль.

$$ESP = ESP - 10_{16} \text{ (} 16_{10} \text{)} = 0018FF5C.$$

```
push 0
push offset str1
push offset str2
push 0
```

Обратите внимание, следующей после

вызова процедуры должна быть команда

POPA(POPAD) с адресом  $EIP = 00401014$  (см. рис. выше). Именно её адрес ( $EIP + \text{длина CALL}$ ) и будет сохранён в стеке для возврата из процедуры MessageBox, а в регистр-указатель следующей команды (EIP) будет помещён адрес первой исполняемой команды вызванной процедуры,  $ESP = ESP - 4 = 0018FF58$ .

| Address  | Hex dump       | Registers (FPU) |
|----------|----------------|-----------------|
| 0040101C | ? FF25 0820400 | JMP DWORD PTR   |
| 00401022 | ? FF25 0020400 | JMP DWORD PTR   |
| 00401028 | ? 0000         | ADD BYTE PTR D  |
| 0040102A | 00             | DB 00           |
| 0040102B | 00             | DB 00           |
| 0040102C | 00             | DB 00           |
| 0040102D | 00             | DB 00           |
| 0040102E | 00             | DB 00           |
| 0040102F | 00             | DB 00           |
| 00401030 | 00             | DB 00           |
| 00401031 | 00             | DB 00           |

| Address  | Hex dump                | Registers (FPU)                         |
|----------|-------------------------|-----------------------------------------|
| 00403000 | 07 09 04 06 01 05 00 00 | EAX 00000000                            |
| 00403008 | 00 00 DD F2 EE 20 E2 E0 | ECX CCCCCCCC                            |
| 00403010 | F8 E0 20 EF E5 F0 E2 E0 | EDX DDDDDDDD                            |
| 00403018 | FF 20 EF F0 EE E3 F0 E0 | EBX BBBBBBBB                            |
| 00403020 | EC EC E0 20 E4 EB FF 20 | ESP 0018FF58                            |
| 00403028 | 57 69 6E 33 32 00 41 73 | EBP 0018FF94                            |
| 00403030 | 73 65 6D 62 6C 65 72 20 | ESI 11111111                            |
| 00403038 | E4 EB FF 20 57 69 6E 64 | EDI FFFFFFFF                            |
| 00403040 | 6F 77 73 20 96 20 FD F2 | EIP 0040101C <JMP. &user32.MessageBoxA> |
| 00403048 | EE 20 F1 EA E0 E7 EA E0 |                                         |
| 00403050 | 21 00 00 00 00 00 00 00 |                                         |
| 00403058 | 00 00 00 00 00 00 00 00 |                                         |

Адрес возврата  
 Передаваемые параметры  
 8 регистров

Для того, чтобы получить доступ к параметрам из стека нужно в начале вызванной процедуры вынуть из стека и сохранить адрес возврата (например командой POP ADR\_RET). Главное – не забыть вернуть адрес возврата в стек перед окончанием процедуры (PUSH ADR\_RET). Затем можно вынимать передаваемые параметры командами POP

```
POP ADR_RET
POP eax      ; ноль
POP ebx      ; адрес str2
POP ecx      ; адрес str1
POP edx      ; ноль
...
PUSH ADR_RET
```

Второй вариант – использовать косвенную адресацию через регистр ESP.

```
MOV eax,[esp+4]      ; ноль
MOV ebx,[esp+8]      ; адрес str2
MOV ecx,[esp+12]     ; адрес str1
MOV edx,[esp+16]     ; ноль
```

По команде RET (возврат из процедуры в вызывающую программу) из вершины стека вынимается адрес возврата и помещается в регистр EIP. Кроме того, по директиве «stdcall» (соглашению, что функция возвращает вершину стека в то состояние, которое было до передачи аргументов), 4 записи с

параметрами функции также аннулируются из стека (если они не были извлечены в процедуре). Теперь в стеке остались только текущие данные программы, что мы сохраняли перед вызовом процедуры – 8 регистров. Их мы и будем вынимать из стека командой POPA в обратном порядке. Как видим, после работы процедуры значение регистров (например, EAX после вызова MessageBox) изменилось, вот зачем мы сохраняли прежние значения в стеке. Можно, конечно сохранить и в дополнительно придуманных переменных ОП, но в стеке – проще. Он ведь специально для этого и существует.

Можно также сохранять и содержимое отдельных регистров и переменных=ячеек памяти (push reg — pop reg или push MEM1 — pop MEM1). Главное восстанавливать значения переменных в правильном порядке – обратном по отношению к порядку сохранения. Стек можно дополнительно использовать как временный буфер для обмена содержимого ячеек памяти (при сортировке). Например, если нужно произвести обмен  $A \leftrightarrow B$ , то можно выполнить следующие действия:

| Вариант 1<br>(через регистры) | Вариант 2<br>(через стек) | Вариант 3 (с использованием<br>команды обмена) |
|-------------------------------|---------------------------|------------------------------------------------|
| mov EAX,A                     | push A                    | mov EAX,A                                      |
| mov EBX,B                     | push B                    | xchg EAX,B                                     |
| mov B,EAX                     | pop A                     | mov A,EAX                                      |
| mov A,EBX                     | pop B                     |                                                |

Процедуры могут быть внутренними и внешними. Если процедура расположена в том же файле, то это внутренняя процедура (SUMMA proc). В рассмотренном фрагменте программы API-функция MessageBox – это внешняя процедура.

## 2. Программирование на Ассемблере MIPS

Для работы с процедурами в Ассемблере MIPS используется несколько соглашений.

- Вызов процедуры осуществляется при помощи инструкции `jal` (**j**ump **a**nd **l**ink), которая делает безусловный переход на адрес начала подпрограммы и сохраняет адрес возврата в 31-й регистр (`$ra`).
- Возврат из процедуры выполняется при помощи инструкции `jr` (**j**ump **r**egister), которая делает безусловный переход по адресу возврата, хранящемуся в 31-м регистре (`$ra`).
- Аргументы в процедуру передаются через регистры `$a0` - `$a3`.
- Возвращаемые значения в вызывающую программу передаются через регистры `$v0`-`$v1`.

### Пример

```
main:
    ...
    addi $a0, $0, 2    # задаем аргумент0 f=2
    addi $a1, $0, 3    # задаем аргумент1 g=3
    addi $a2, $0, 4    # задаем аргумент2 h=4
    addi $a3, $0, 5    # задаем аргумент3 i=5
    jal  diffofsums    # вызываем процедуру
    add  $s0, $v0, $0   # возвращаемое значение из $v0
    ...

diffofsums:           # $s0 = result
    add $t0, $a0, $a1  # $t0 = f + g
    add $t1, $a2, $a3  # $t1 = h + i
    sub $s0, $t0, $t1  # result = (f + g) - (h + i)
    add $v0, $s0, $0   # записать возвращаемое значение в $v0
    jr  $ra           # возвращаемся в вызывающую программу
```

Данные также можно передать через стек. Регистр указателя стека `$sp` (**s**tack **p**ointer) содержит адрес вершины стека. По мере расширения стека его вершина сдвигается в сторону младших адресов.

```
.text
addi $sp, $sp, -16    # выделяем место в стеке для сохранения 4-х аргументов
addi $a0, $0, 2      # задаем аргумент0 f=2
sw   $a0, 12($sp)    # сохраняем в стеке регистр $a0=аргумент2=f
addi $a0, $0, 3      # задаем аргумент1 g=3
sw   $a0, 8($sp)     # сохраняем в стеке аргумент1 g=3
addi $a0, $0, 4      # задаем аргумент2 h=4
sw   $a0, 4($sp)     # сохраняем в стеке аргумент2 h=4
addi $a0, $0, 5      # задаем аргумент3 i=5
```



```

sw    $a0,0($sp)    # сохраняем в стеке аргумент3 i=5
jal   diffofsums    # вызываем процедуру
li    $v0,1          # определяем сервис вывода целого
syscall
j     stop_prog

```

```

diffofsums:          # результат в $a0 для вывода переменной через syscall
lw    $v0,4($sp)    # восстанавливаем из стека аргумент2 h=4
lw    $a0,0($sp)    # восстанавливаем из стека аргумент3 i=5
add   $v0,$v0,$a0    # $v0 = h + i
lw    $a0,8($sp)    # восстанавливаем из стека аргумент1 g=3
lw    $v1,12($sp)   # сохраняем в стеке аргумент2=f
add   $v1,$v1,$a0    # $v1 = f + g
sub   $a0,$v1,$v0    # result = (f + g) - (h + i)
addi $sp,$sp,16     # очищаем место в стеке
jr    $ra            # возвращаемся в вызывающую программу

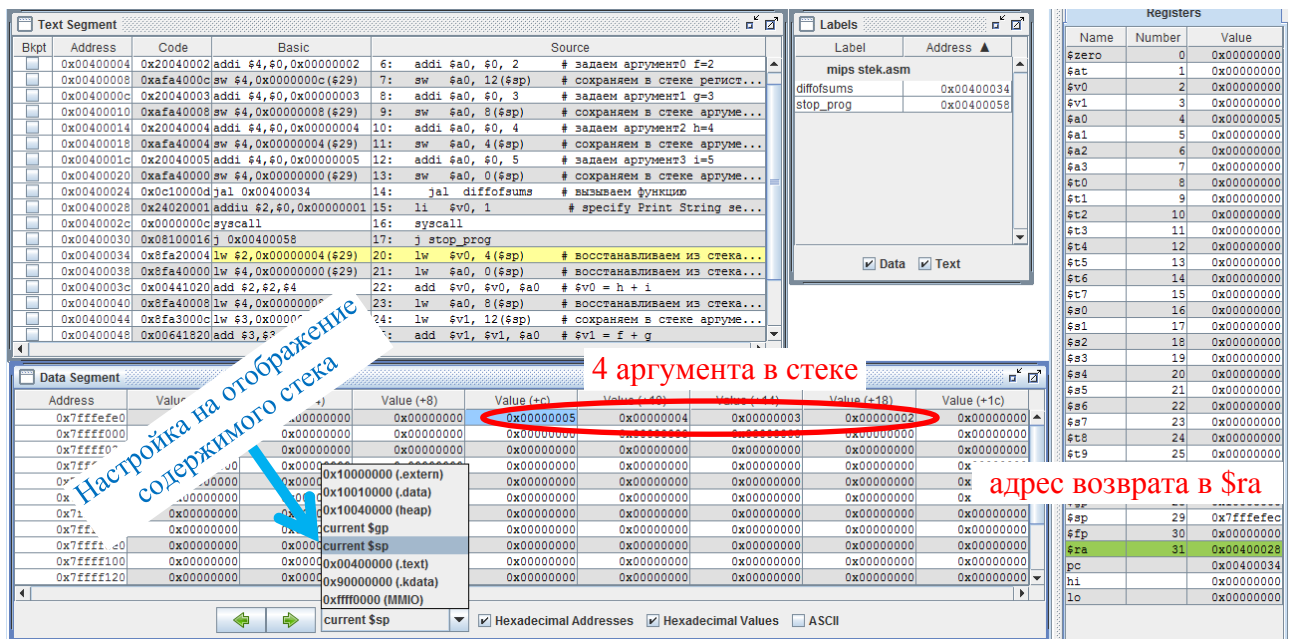
```

```

stop_prog:
nop

```

Если настроить дамп памяти на отображение содержимого стека, то можно увидеть, как в стек помещаются все передаваемые параметры.



Кроме того, для корректной работы с общими для всех программ/процедур регистрами нужно в самом начале вызываемой процедуры сохранять в стеке текущие значения регистров, которые будут изменены ей.

Если сама вызываемая процедура (proc1) также требует вызова дополнительной вложенной процедуры (proc2), то в стеке дополнительно

сохраняется текущий адрес возврата к вызывающей программе предыдущего уровня вложенности (main).

```

proc1:
  addi $sp,$sp,-4 # выделяем место в стеке
  sw   $ra,0($sp) # сохраняем регистр $ra – адрес возврата в main
      jal  proc2   # вызываем proc2 (запоминая в $ra адрес возврата в proc1)
  ...
  lw   $ra,0($sp) # восстанавливаем регистр $ra
  addi $sp,$sp,4  # восстанавливаем размер стека
  jr   $ra        # возврат в вызывающую программу main

```

Итак, отметим корректное программирование вызовов подпрограмм.

### Вызывающая программа

Сохраняет в стеке регистры, которые может модифицировать вызываемая процедура (например: \$t0-t9)

Записывает аргументы вызываемой процедуры в регистры \$a0-\$a3

Делает переход jal на вызываемую процедуру

После возврата из вызываемой процедуры считывает результат из \$v0

Восстанавливает необходимые регистры

### Вызываемая процедура

Сохраняет регистры, которые планирует использовать (например: \$ra, \$s0-\$s7)

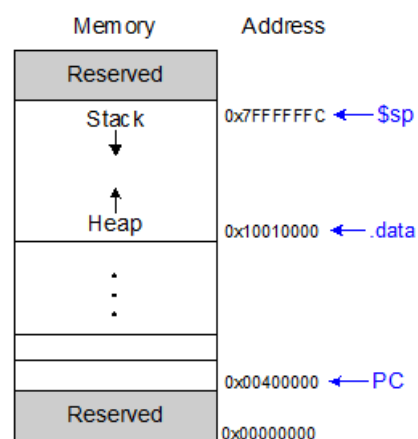
Выполняет необходимые операции

Записывает результат в \$v0

Восстанавливает сохраненные регистры

Выполняет возврат в вызывающую программу (jr \$ra)

В архитектуре MIPS карта памяти выглядит следующим образом. Как видно область сегмента данных (.data) и сегмента стека размещены близко друг к другу и заполняются во встречных направлениях.



Программист должен самостоятельно следить, чтобы эти сегменты не нарушили границы друг друга.

## 2. ЗАДАНИЕ

1. В любой, написанной вами программе к одной из ПР № 2,3,4 оформить изменения данных, или вывод сообщений, или часть кода в виде внутренней процедуры с передачей параметров через стек.
2. Ответить на вопросы для самопроверки.

## 3. КОНТРОЛЬНЫЕ ВОПРОСЫ

17. Расскажите об изменениях в регистрах процессора и ячейках ОП по ходу выполнения вашей программы (Intel и MIPS).
18. Что такое процедуры?
19. Как обозначаются процедуры в программе на Ассемблере (Intel и MIPS)?
20. Как происходит вызов процедур в программе на Ассемблере (Intel и MIPS)?
21. Как передаются параметры в процедуры (Intel и MIPS)?
22. Как меняется содержимое регистров IP/PC, SP и стека при вызове процедур (Intel и MIPS)?

## 4. СПИСОК ЛИТЕРАТУРЫ

1. Рикардо Нарваха. Введение в крэкинг с нуля, используя OllyDbg. Глава 1 (<http://pro.dtn.ru/cr.html>).
2. Intel 64 and IA32-Architectures Software Developers Manual (с сайта intel.com)
3. Гук М., Юров В. Процессоры Pentium III, Athlon и другие. – СПб.: Издательство «Питер», 2000.
4. Юров В. Ассемблер. – СПб.: Издательство «Питер», 2002.
5. Хэррис, Д. М. Цифровая схемотехника и архитектура компьютера, 2017г (2019)