**МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»**

**ИНСТИТУТ ТРАНСПОРТНОЙ ТЕХНИКИ И СИСТЕМ**

**УПРАВЛЕНИЯ (ИТТСУ)**

**Кафедра «Управление и защита информации»**

**МАШИННО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЯЗЫК АССЕМБЛЕРА (ДЛЯ INTEL P6)**

Методические указания к лабораторным работам

по дисциплине «Машинно-ориентированные языки программирования»

Москва – 2021

У т в е р ж д е н о редакционно-издательским советом университета

**МАШИННО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЯЗЫК АССЕМБЛЕРА (ДЛЯ INTEL P6)**

Москва – 2021

УДК: 002: 621. 331: 621. 311: 681. 3 B24

**Гречишников В.А., Логинова Л.Н. Машинно-ориентированные языки программирования. Язык ассемблера (для Intel P6).**: Методические указания к лабораторным работам. – М.: МИИТ, 2021. – 57 с.

** Российский университет транспорта (МИИТ), 2021**

**Содержание**

[ВВЕДЕНИЕ 4](#_bookmark0)

[Порядок выполнения лабораторных работ 8](#_bookmark1)

[Лабораторная работа №1. Регистры, данные и команды пересылки данных 18](#_bookmark2)

[Лабораторная работа №2. Арифметические команды 29](#_bookmark3)

[Лабораторная работа №3. Команды передачи управления 39](#_bookmark4)

[Лабораторная работа №4. Логические команды и команды манипулирования битами 47](#_bookmark5)

[Список использованной и рекомендуемой литературы 55](#_bookmark6)

# ВВЕДЕНИЕ

После создания языков программирования высокого уровня, т.е. таких языков программирования, которые несколько приближенны к языку и мышлению человека (таких, как *Fortran*, *Basic*, *Pascal*, *C++* и их современные объектно-ориентированные продолжатели *Visual Basic*, *Delphi*, *C#* и т.д.) постоянно задается вопрос, а зачем надо изучать язык Ассемблера, относящийся к машинным, низкоуровневым языкам? С тех же времен на этот вопрос дается традиционный ответ, который не только не утратил своей актуальности, но и может быть расширен.

На сегодняшний день промышленностью разных стран выпускается большое количество микропроцессоров и микроконтроллеров, которые представляют собой интегральные микросхемы. Микроконтроллер – это самостоятельный класс устройств, который содержит в себе, в едином корпусе, микропроцессорное ядро и набор периферийного оборудования, такого как таймер/счетчики, каналы ввода/вывода, широтно-импульсные модуляторы, аналого-цифровые и цифро-аналоговые преобразователи, сторожевой таймер, менеджер питания, тепловой сенсор и т.д., поэтому далее при разговоре про ассемблер будем употреблять слово микропроцессор, подразумевая под ним и микроконтроллер с его микропроцессорным ядром. И микропроцессор, и микроконтроллер представляют собой конечные автоматы с программным принципом управления, а значит, последовательность двоичных кодов позволяет управлять внутренним состоянием микропроцессора и микроконтроллера и всеми внутренними компонентами, заставляя их, в конечном итоге, приходить к состоянию, необходимому в данной задаче. Последовательность двоичных кодов и есть программа для микропроцессора или программа на машинном языке. Язык ассемблера – это переведенный на псевдочеловеческий язык машинный код. Т.е. ассемблер теснейшим образом связан с внутренним устройством, архитектурой и представлением информации в микропроцессоре. Значит, если программист или инженер микропроцессорной техники знает язык ассемблера, то он однозначно знает микропроцессор, а значит, может напрямую взаимодействовать с ним, заставляя микропроцессор эффективнее решать требуемые задачи. По аналогии можно сказать, что если Вы знаете какой-либо иностранный язык, то Вы можете эффективнее жить в государстве, где данный иностранный язык государственный, и не пользоваться переводчиками (людьми, разговорниками, электронными переводчиками), которые не всегда точно могут передать Вашу мысль собеседнику. Хотелось бы отметить, что языков Ассемблера множество, так как множество архитектур построения микропроцессоров. Фактически, для каждого семейства микропроцессоров есть свой ассемблер.

Зная о том, что язык Ассемблера – это машинный язык, а значит, с помощью его можно эффективнее управлять микропроцессором, профессиональные программисты используют его в самых критических по времени местах программ, даже если программируют на языках высокого уровня. Например, если посмотреть модуль *System* в *Delphi*, который выполняет

все базовые функции и автоматически подключается ко всем проектам, то в нем можно увидеть множество ассемблерных вставок. Это понятно, так как, если этот модуль основной (т.е. в нем содержаться основные процедуры и функции, которые чаще всего выполняются), то нужно, чтобы они быстро и эффективно обрабатывались микропроцессором, а значит, их и реализуют на ассемблере. Все основные математические функции написаны на языке Ассемблере. Вот, например, как выглядит реализация функции вычисления экспоненты:

function Exp(const X: Extended): Extended; asm

end;

FLD X FLDL2E FMUL

FLD ST(0) FRNDINT

FSUB ST(1), ST FXCH ST(1) F2XM1

FLD1 FADD FSCALE

FSTP ST(1)

Также, все, что касается работы с памятью, тоже реализуется с помощью

ассемблерных вставок, так как работа с памятью происходит постоянно, то здесь необходима самая высокая скорость и эффективность. Желающие могут посмотреть реализацию функции *Move* в модуле *System*, а также все, что касается реализации менеджера памяти.

Некоторые могут заметить, что компилятор и так занимается тем, что переводит с языка высокого уровня на машинный, а значит, и так понятно, что в конечном итоге все будет приведено к машинному коду. Это так, однако, ассемблерные вставки все равно используют. Почему? Дело в том, что язык высокого уровня строится на некотором базовом наборе функциональных возможностей или кирпичиков, с помощью которых реализуются все дальнейшие программные конструкции любой сложности. Поскольку набор таких кирпичиков, хоть и увеличивается с развитием языка, но до сих пор ограничен и обладает некоторым универсализмом, позволяющим использовать их в любых ситуациях и допустимых сочетаниях. Это приводит к тому, что при компилировании для каждого кирпичика формируется универсальный ассемблерный блок, который будет пригоден для любых ситуаций, но обладает некоторой избыточностью, как плата за универсальность, а значит, менее эффективен. В современных языках высокого уровня избыточность выходного машинного кода небольшая и ее стараются постоянно уменьшить, но избавиться совсем от нее невозможно.

Мы уже отмечали, что знание ассемблера означает знание микропроцессора, а каждый микропроцессор обладает своими особенностями.

Быстродействие каждого микропроцессора зависит не только от тактовой частоты и объема кеш-памяти. На сегодняшний день разработано большое количество инженерных ухищрений, которые позволяют повысить быстродействие микропроцессора. К таким ухищрениям относятся суперскалярная конвейерная архитектура, блок предсказания ветвления, несколько исполнительных блоков (целочисленные вычисления, вычисления с плавающей точкой, выполнение инструкций мультимедийного расширения, потоковых команд и т.д.), внутренняя переадресация регистров, изменение очередности выполнения микрокоманд и т.д. Ситуация усложняется выходом на рынок многоядерных микропроцессоров, где архитектура еще сложнее. Все это приводит к тому, что реализация одних и тех же алгоритмов разными наборами машинных кодов на одном микропроцессоре, так же как и реализация одних и тех же алгоритмов одними и теми же наборами команд на разных микропроцессорах одного класса при прочих равных условиях, будут выполняться с разными скоростями! Это происходит из-за того, например, что внутренняя архитектура одного микропроцессора на одном ядре позволяет выполнять параллельно несколько команд в определенном сочетании, а у другого микропроцессора не позволяет. При этом, языки высокого уровня разрабатываются универсальными для разных типов процессоров, а значит, редко используют внутренние особенности архитектуры микропроцессоров, а значит, могут создавать не очень эффективный код для любых микропроцессоров или для одного более эффективный, а для другого менее эффективный. Если же программировать на ассемблере под конкретный микропроцессор, зная все особенности его внутреннего построения, то машинный код можно будет всегда сделать эффективнее.

Если речь идет о встраиваемых микропроцессорных системах, мобильных устройствах, промышленных компьютерах, интеллектуальных микропроцессорных датчиках, измерителях и регистраторах и т.д., то вопрос об эффективности многократно возрастает, так как все подобные системы для уменьшения стоимости, энергопотребления, габаритных показателей производятся с ограниченными объемами памяти и производительностью, но всегда функционально загруженными «под завязку», да еще и есть желание расширить его функциональный состав. Программирование таких систем, в которых необходимо выполнять математические вычисления, контролировать работу всех периферийных устройств, принимать и передавать информацию и т.д., не обходится и не может обойтись без использования ассемблера.

Некоторые могут заметить, что они не собираются быть профессиональными программистами, тем более программировать мобильные системы или промышленные микропроцессорные контроллеры, а значит, знания об ассемблере им не нужны. Да, с этим можно отчасти согласиться. Но Вас никто на профессиональных программистов не готовит. Вы обучаетесь в университете, а значит, получаете академическое инженерное образование с широким кругозором в лучших традициях советской высшей школы. На сегодняшний день человечество вступило в век информации, а подавляющее большинство информационных процессов зиждется на цифровых технологиях с

применением микропроцессорной техники, и теперь уже сложно представить рабочее место человека без персонального компьютера, а самого человека без мобильного телефона. Значит, современный инженер должен понимать основы функционирования современных микропроцессорных систем, знать их лучше, чем обыватель, ибо просто работать на компьютере может уже и первоклассник. А, поскольку, функционирование микропроцессорной техники основано на программном принципе управления, поскольку есть множество принципов построения архитектур микропроцессоров и микропроцессорных систем, поскольку есть множество видов представления информации в микропроцессорных системах (а знания о представлении информации позволяет более глубоко понимать работу всех вычислительных и не только программ, таких как *MS Excel*, СУБД, программы обработки изображений, векторной графики, инженерных программ, например, *MatLab*, *MathCad*, все языки программирования и т.д.), то знание низкоуровневого языка программирования ассемблера, изучение которого теснейшим образом связано с обозначенными вопросами, несомненно, является *обязательным!*

# Порядок выполнения лабораторных работ

Для изучения основ языка Ассемблера был выбран микропроцессор семейства *Intel P6*, как самый распространенный и широкодоступный микропроцессор, так как он чаще всего устанавливается в современных персональных компьютерах. Как уже упоминалось во введении, языков ассемблера много. Изучая ассемблер для одного из передовых, мощных, современных микропроцессоров, какими являются процессоры семейства *Intel P6*, автоматически охватываются все процессоры семейств *x86, IA32*, на уровне мнемоники *MCS-51* и т.д., и формируются фундамент для быстрого изучения любого ассемблера и микропроцессора.

Средой для изучения языка Ассемблера выбрана интегрированная среда разработки приложений *Visual Studio*, основанная на языке *С++* и возможности делать в тексте программы ассемблерные вставки. Такой подход видится предпочтительным, так как язык *С++* изучается с первого курса, нет необходимости заниматься программированием интерфейса для ввода и вывода данных в программе на ассемблере, наличие встроенного отладчика на уровне *CPU* и т.д.

Чтобы сосредоточиться на изучении именно языка ассемблера и не терять каждый раз время на подготовку формы главного окна проекта и небольшого сервисного кода, создадим единожды заготовку такого приложения и сохраним его в папку.

На первом шаге создадим новый проект в *Visual Studio*. Для этого, после запуска *Visual Studio*, последовательно выбираем «*Создание проекта*» (см.рис.1)

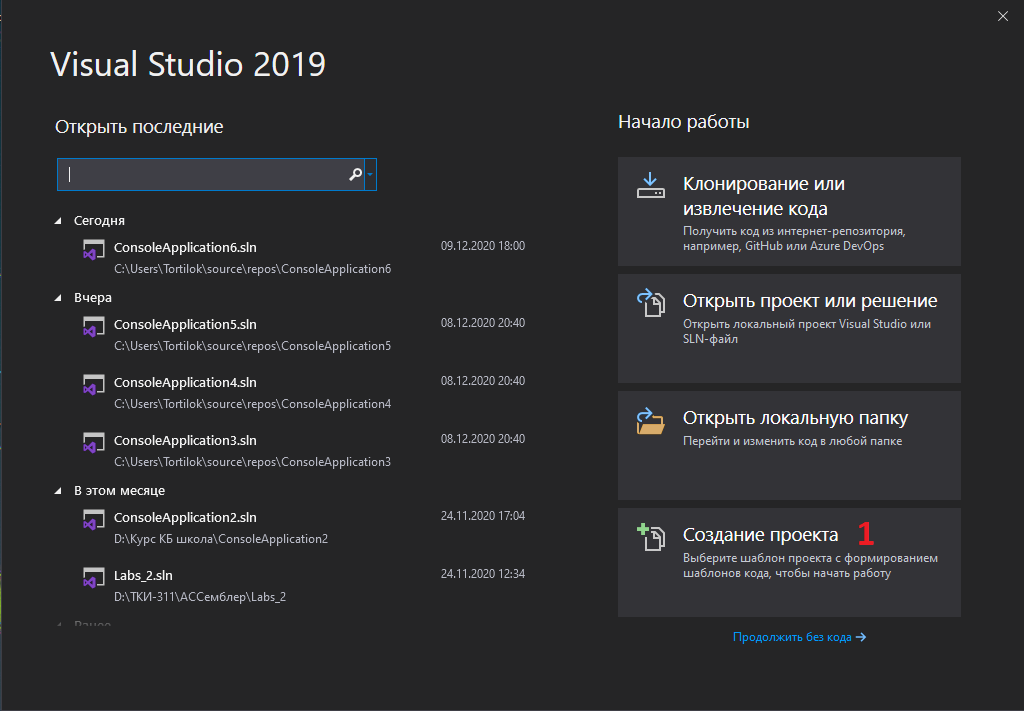


Рисунок 1. Создание проекта.

После выбора «*Создание проекта*» в открывшемся окне выбираем тип нашего приложения, в наших лабораторных работах нужно выбрать пункт «*Консольное приложение»* (см. рис. 2)

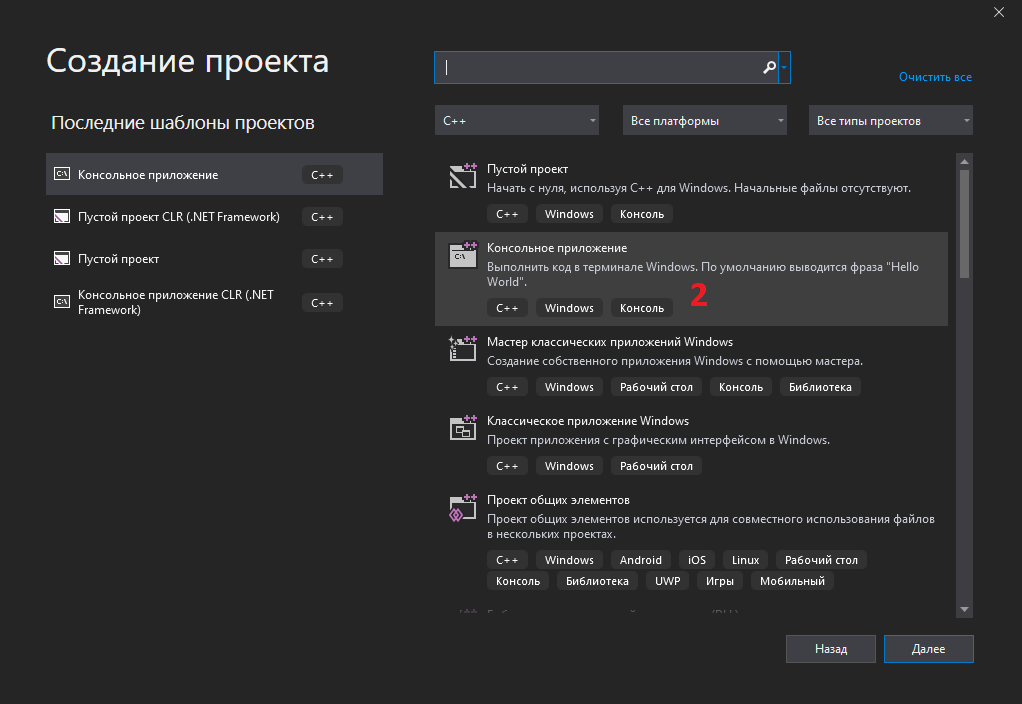


Рисунок 2. Выбор типа проекта.

Далее открывается меню по настройке нашего проекта (см. рис. 3), где

1. пункт отвечает за название нашего проекта,
2. за его расположение в компьютере,
3. поле «*Поместить решение и проект в одном каталоге*», где необходимо поставить галочку.

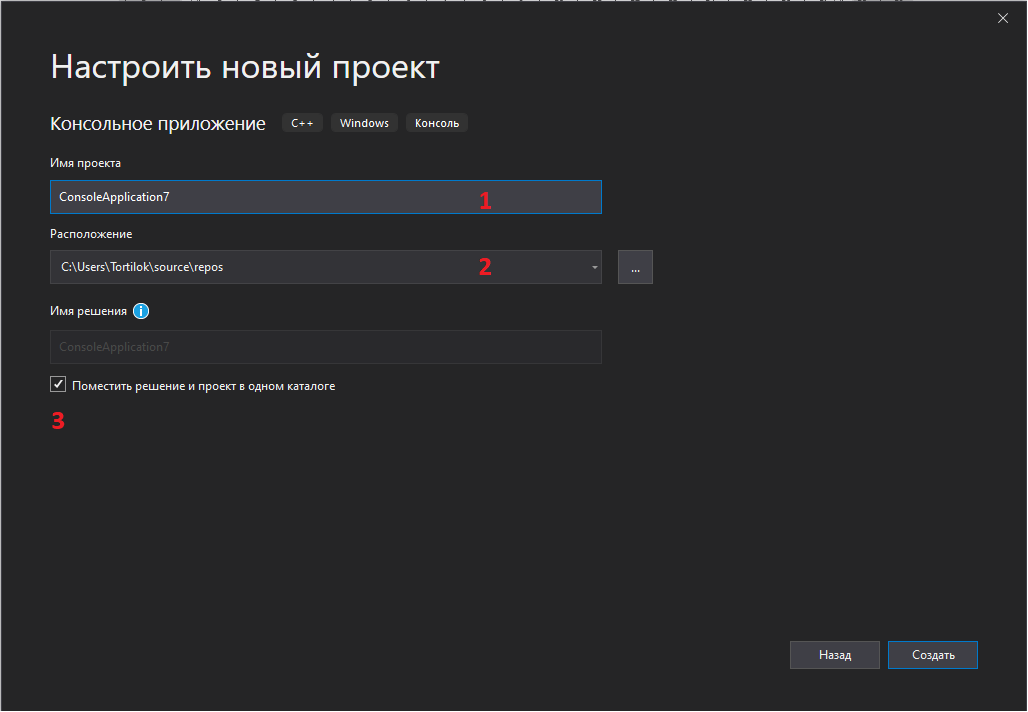


Рисунок 3. Настройка проекта, где 1 – Имя проекта; 2 – Расположение проекта в ПК; 3 – Пункт для выбора.

После нажатия на кнопку «Создать» (см. рис.3) будет создан проект в *Visual Studio* (см. рис. 4).

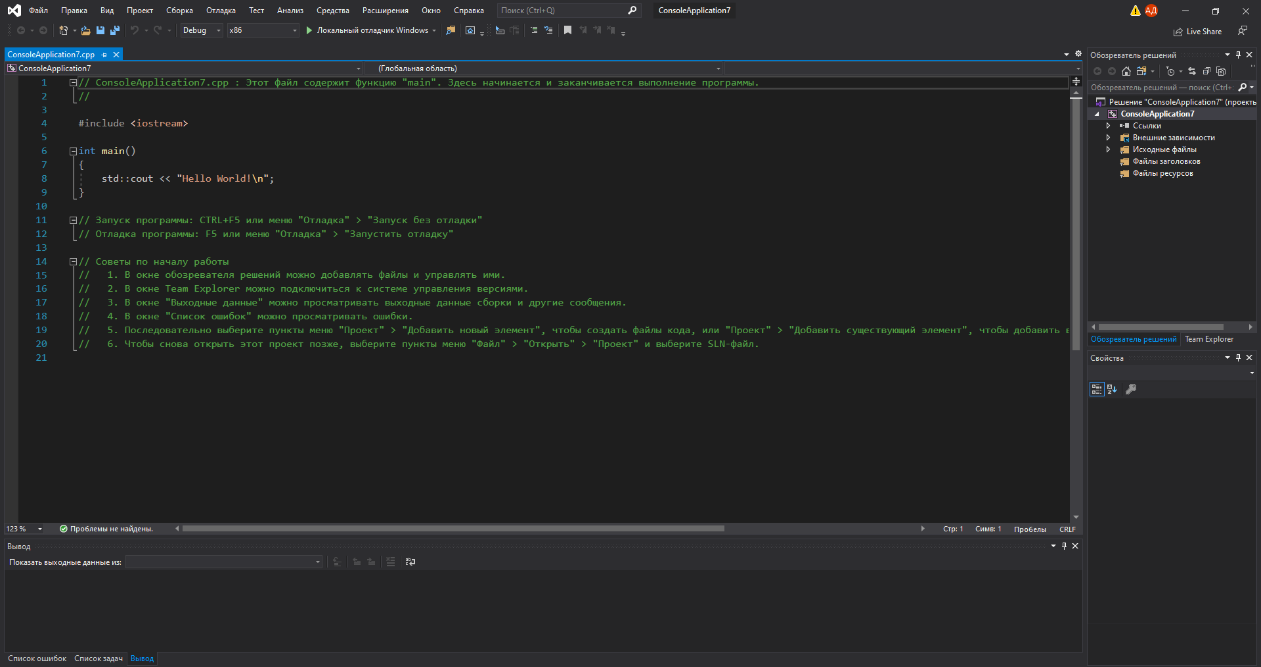


Рисунок 4. Наш проект в Visual Studio.

Ниже расположен макет кода, который необходимо реализовать в проекте. В макете определяется массив *Data*, который необходим для того, чтобы у ассемблерной программы, написанной на языке Ассемблера была область оперативной памяти для свободного доступа и использования. Дело в том, что операционная система (ОС) *Windows*, работая в защищенном режиме, контролирует использование памяти, запрещая доступ в области, куда загружены системные программы. В общем случае контролируется доступ в программные области других запущенных программ. В связи с этим нужно позаботиться о том, чтобы *Windows* выделил программе область памяти, которую можно использовать без запрета доступа. Самым простым способом является объявление массива необходимой длины, каким и является массив *Data*.

После того, как все обработчики событий созданы, проект необходимо скомпилировать, последовательно выбрав в главном меню «*Отладка*→*Начать отладку*» или нажать <*F5>.* Если получены ошибки, то необходимо исправить их и повторно провести компиляцию

Если ошибок нет, то заготовка проекта для всех лабораторных работ готова и теперь ее надо сохранить в репозиторий.

#include<iostream>

#include<conio.h>

#include<iomanip>

#include<sstream>

using namespace std;

const int DataSize = 144;

std::string IntToHex(int n)

{

std::stringstream ss;

ss << std::hex << n;

return ss.str();

}

int main()

{

unsigned char Memo[DataSize];

for (int i = 0; i++; i < DataSize) {

Memo[i] = NULL;

}

\_\_asm {

pushad

//Ваш код на языке ассемблера

popad

}

for (int i = 0; i < DataSize; i++) {

if ((i % 16) == 0) {

cout << "\n" << setw(2) << i / 16 << ":";

}

else {

cout << " " << setw(2) << IntToHex(Memo[i - 1]);

}

}

\_getch();

return 0;

}

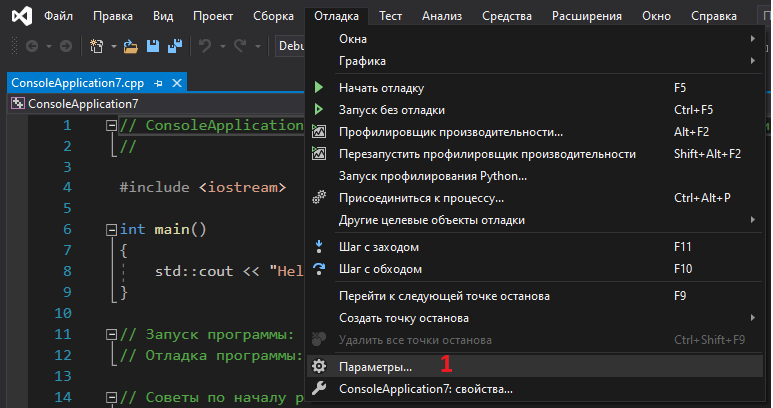
Далее необходимо подготовим проект для работы с ассемблерными ставками. Для этого необходимо перехоти в пункт «*Отладка*→*Параметры*». (см. рис. 5)

Рисунок 5. Меню откладки.

В появившемся окне «*Параметры*» необходимо поставить галочку в пункте «*Включить отладку на уровне адреса*». (см. рис. 6)

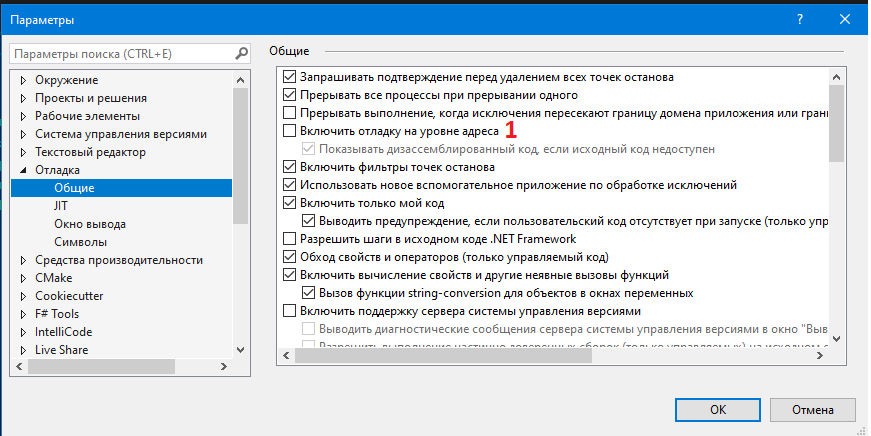


Рисунок 6. Параметры

Далее требуется установить точку останова на первой команде ассемблерной вставки. Для этого необходимо поместить указатель мыши на сером поле слева напротив первой команды ассемблерной вставки ***\_asm{*** и нажать левую кнопку мыши. В результате на сером поле будет отображена красная точка (см. рис. 7).

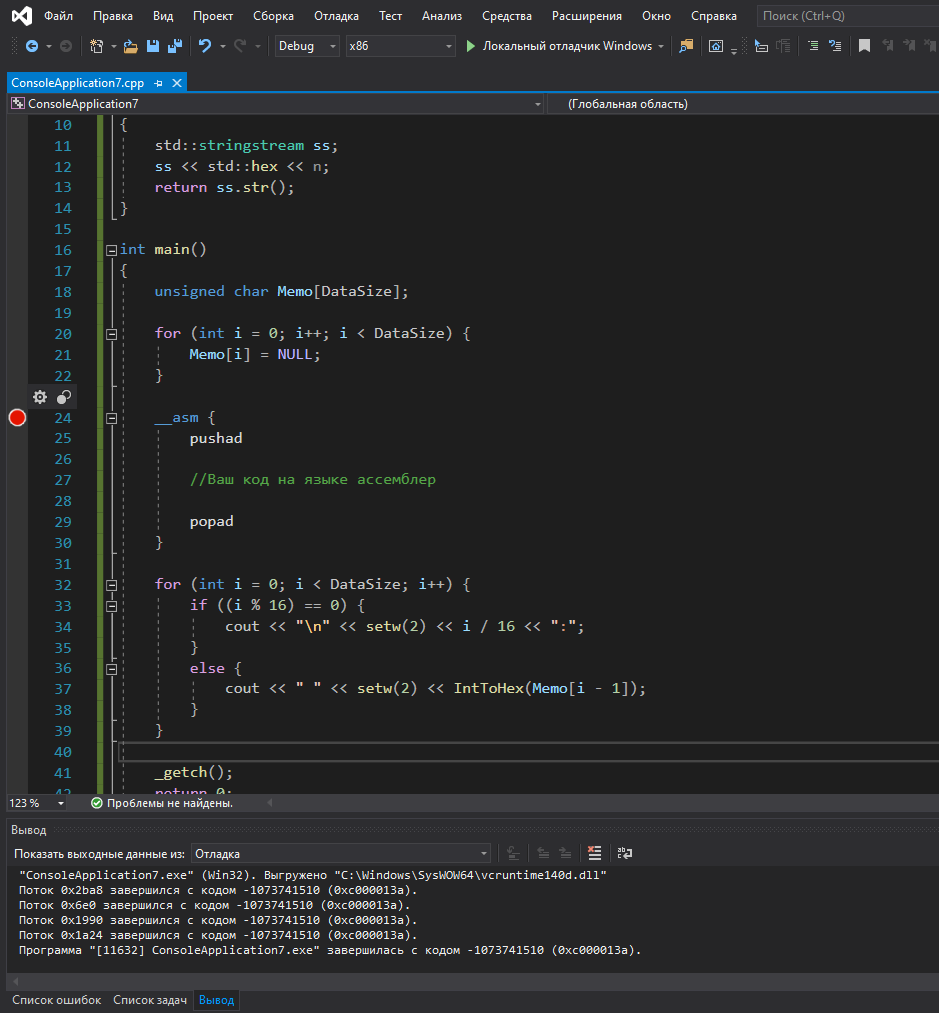


Рисунок 7. Установка точки остановки в проекте

Далее следует запустить отладку программы. После этого необходимо перейти в пункт меню «*Отладка* → *Окна* → *Дизассемблированный код*», после чего будет отображен дизассемблированный код программы (см. рис. 8 - 9).

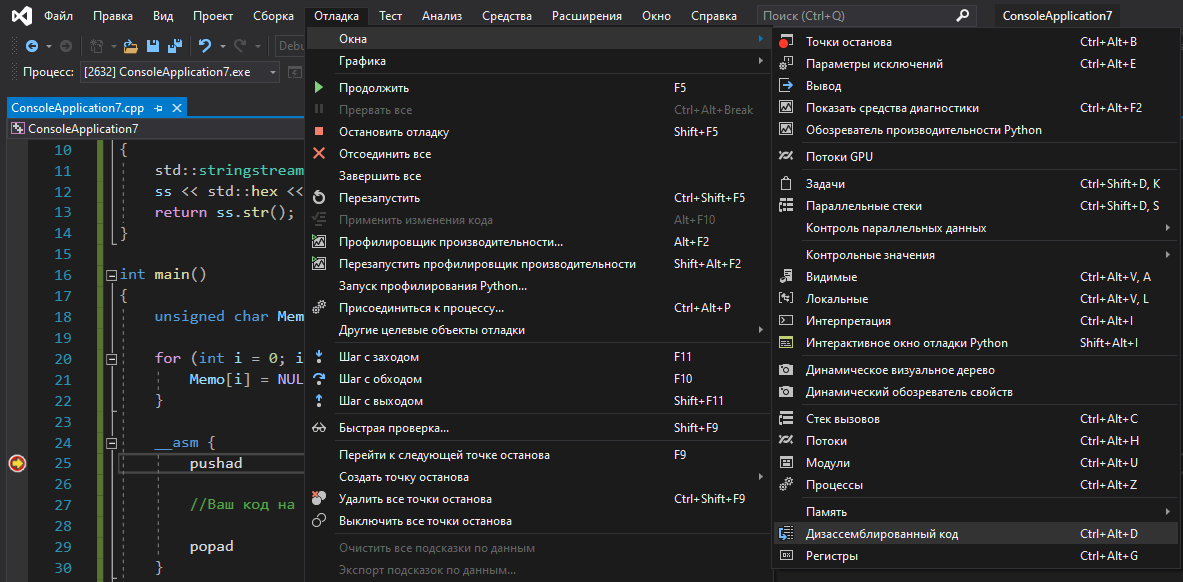


Рисунок 8. Открытие дизассемблированного кода

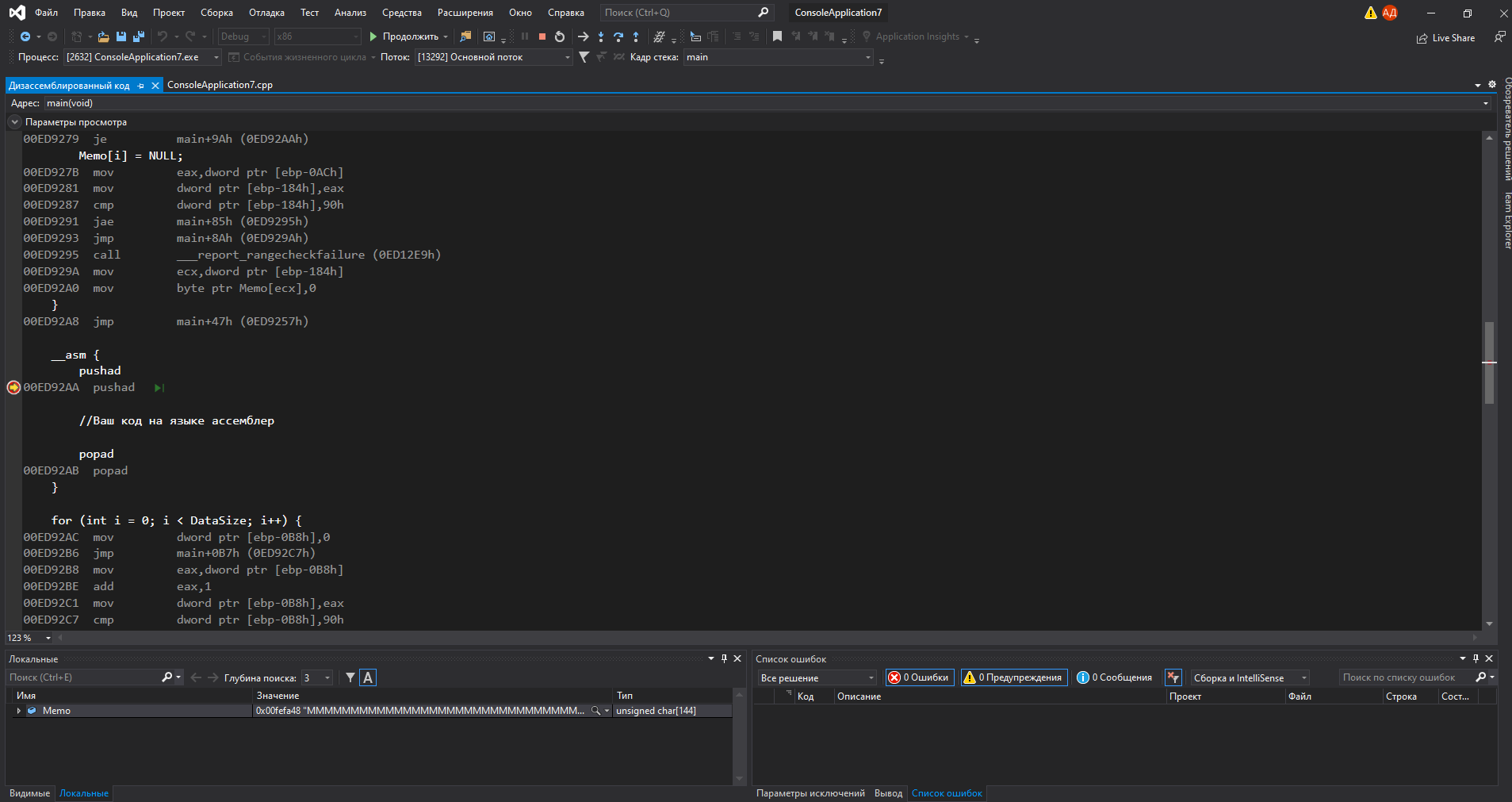


Рисунок 9. Дизассемблированный код

На рисунке 9 первая колонка – адрес команды в сегменте кода, вторая колонка – текст на языке ассемблера.

Выделенные жирным шрифтом строчки – строчки программы на *C++*, а строчки, отображаемые обычным шрифтом, расположенные под ней – соответствующий ей ассемблерный код. Красная точка слева от первого столбца – точка останова. Еще левее отображается желтая стрелка – указатель на команду, которая будет исполнена.

При необходимости можно отобразить Окна с содержимым регистров общего и специального назначения. Для этого следует воспользоваться пунктом меню «*Отладка* → *Окна* → *Регистры*» (см. рис. 10).

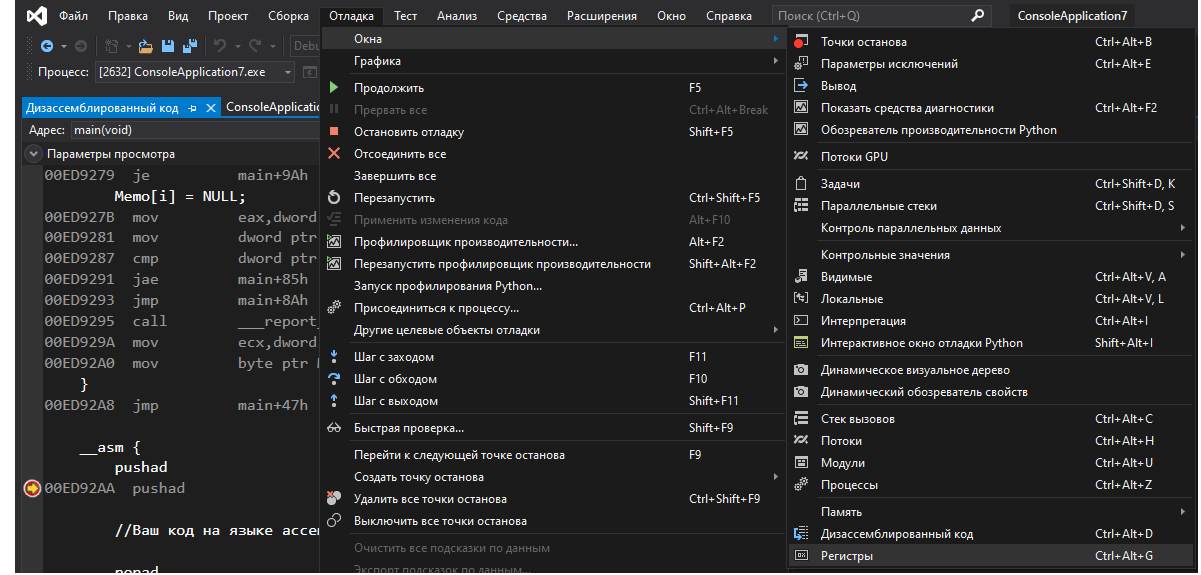


Рисунок 10. Открытие регистров.

Окно регистров можно перемещать по экрану и выбрать требуемое место для расположения (см. рис. 11).

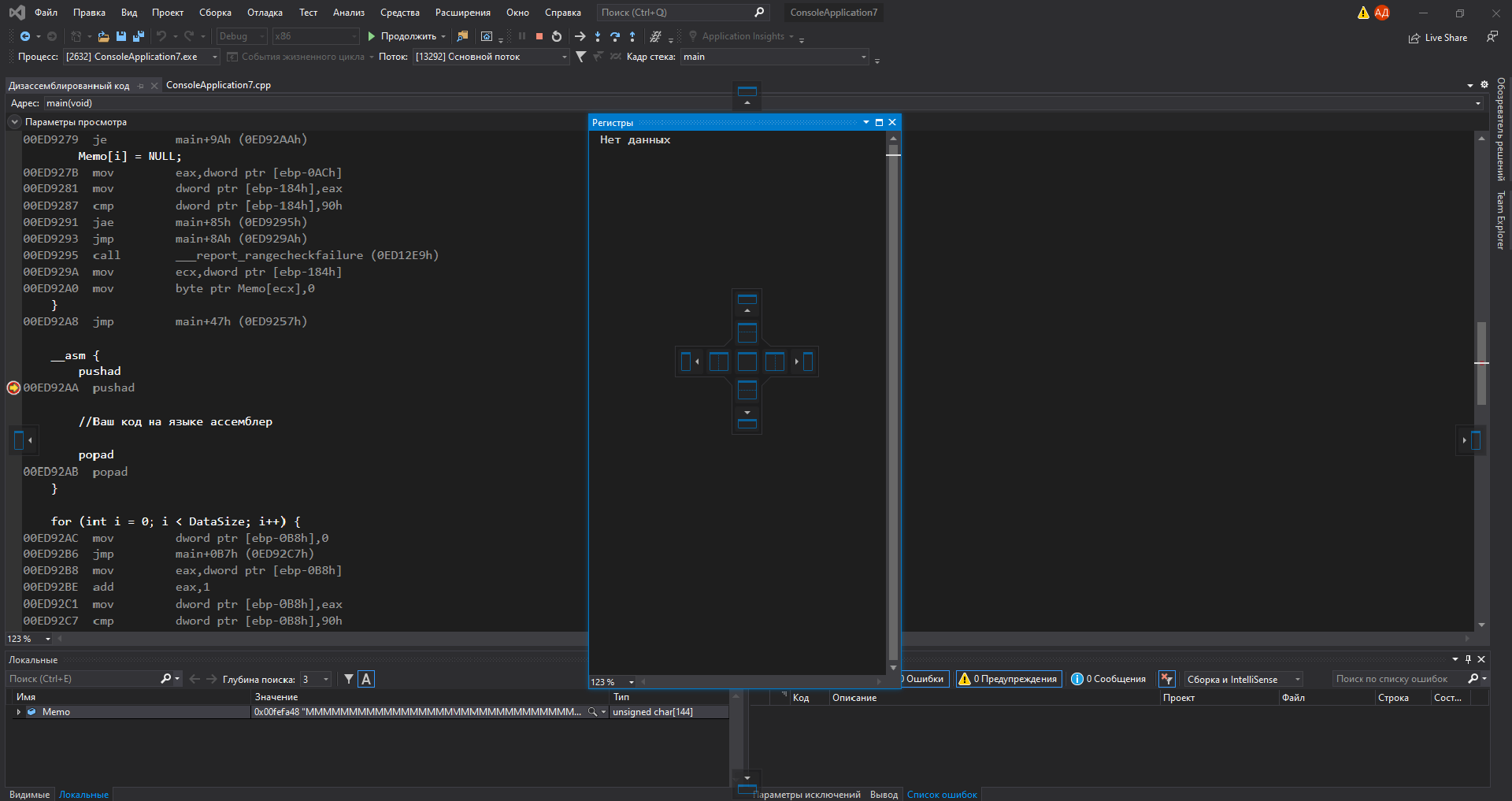


Рисунок 11. Выбор места расположения окна регистров

После выбора требуемого места расположения необходимо нажать по окну правой кнопкой мыши и в результате будет отображено контекстное меню с выбором различных регистров, сегментов центрального процессора (ЦП), флагов. Следует выбрать «*ЦП*», «*Флаги*» (см. рис. 12, 13).

|  |  |
| --- | --- |
| Рисунок 12. Меню выбора регистров, сегмента ЦП, Флагов | Рисунок 13. Отображение регистров общего назначения и флагов процессора |

Далее отобразим содержимое оперативной памяти, соответствующее сегменту данных, для этого необходимо перейти в пункт меню «*Отладка* → *Окна* → *Память* → *Память2*» (см. рис. 14).

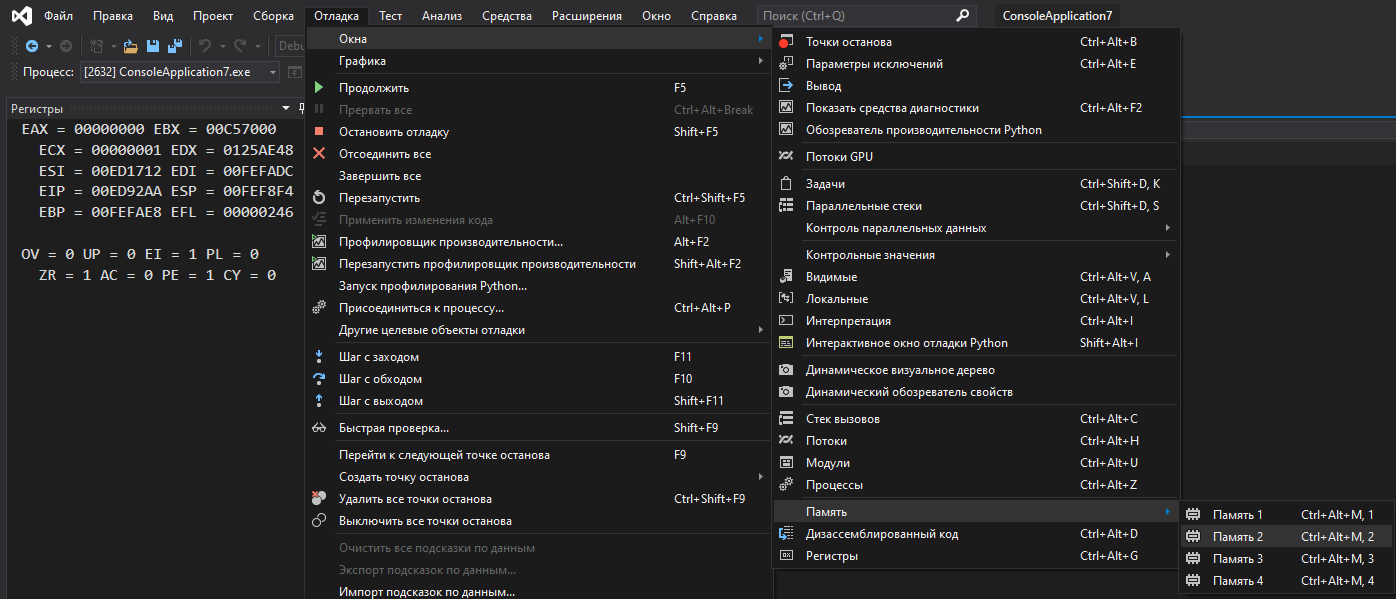


Рисунок 14. Открытие памяти

После появления окна следует выбрать место для расположения (см. рис. 15).

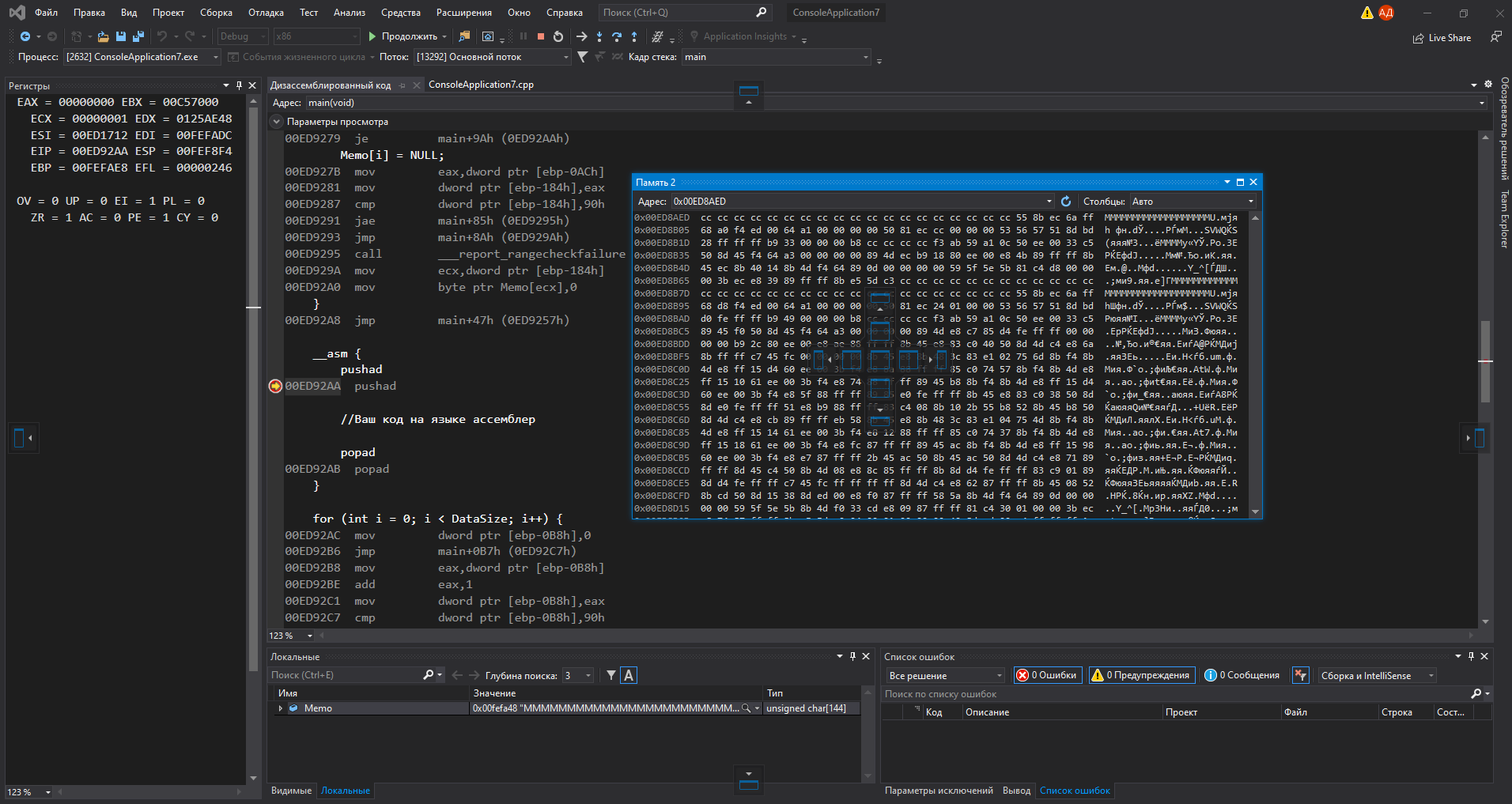


Рисунок 15. Выбор расположения окна памяти

Память 2 служит для отображения содержимого памяти в сегменте данных в виде, так называемого, дампа памяти. В первой колонке выводиться адрес первой ячейки памяти, отображаемой в строке. Во втором столбце отображается информация о значениях ячеек памяти. В третьем столбце отображается код *ASCII* представление каждого байта в строке (см. рис. 16).

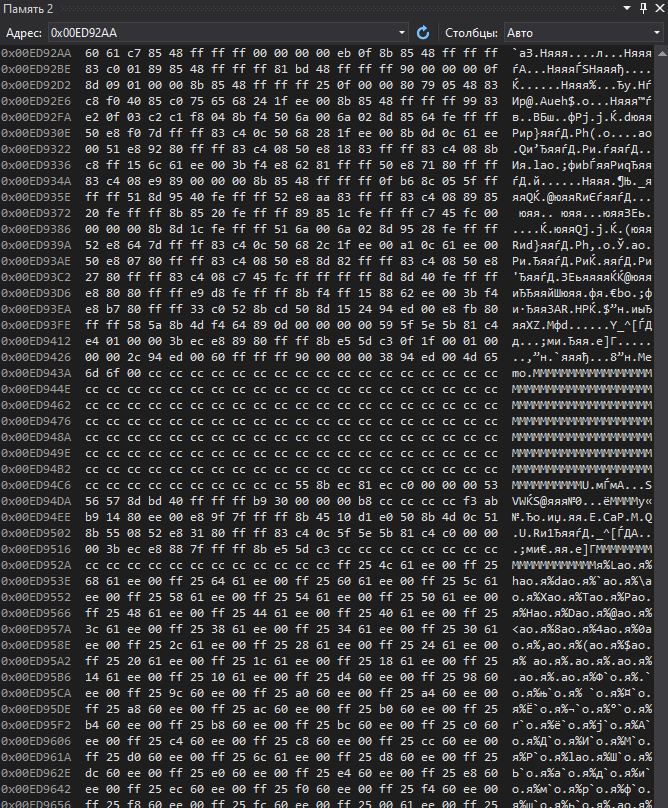


Рисунок 16. Дамп памяти

Изменять отображение данных можно с помощью контекстного меню, которое вызывается нажатием Правой кнопки мыши (см. рис. 17), при этом

Пункт 1 – служит для изменения начального адреса памяти для отображения, Пункт 2 – выбор количества отображаемых ячеек памяти,

Пункт 3 – меню для изменения отображения данных.

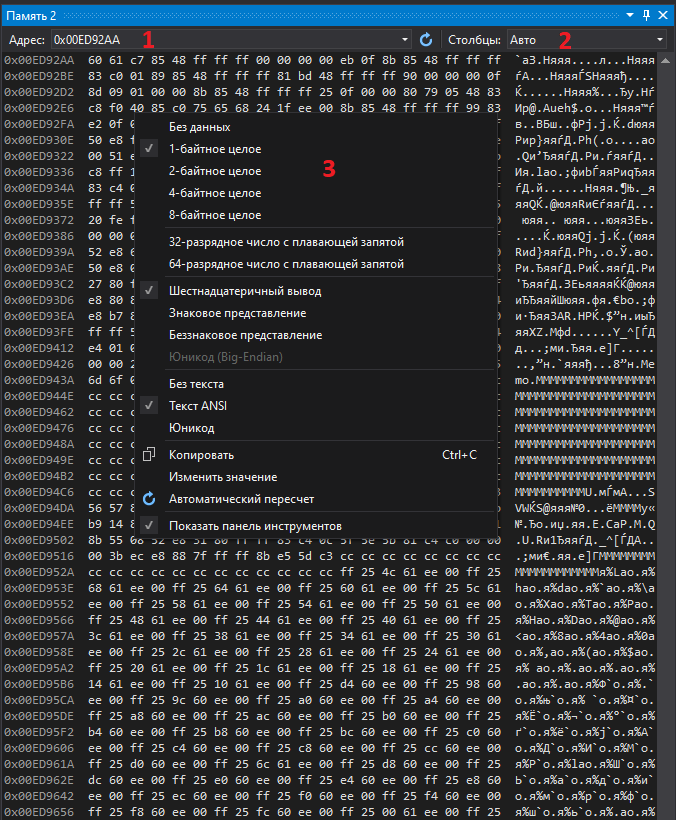


Рисунок 17. Настройка отображения дампа памяти.

Пункт 1 – ссылка на адрес;

Пункт 2 – количество отображаемых ячеек памяти;

Пункт 3 – меню для смены отображения данных

После размещения окна памяти и его настройки проект *Visual Studio* готов для работы с ассемблерными вставками. Приведенную выше процедуру следует выполнить единожды, IDE запомнит ваши настройки автоматически. Проект следует сохранить (см. рис. 18).

Работа с новым проектом проходит в следующем порядке:

1. Необходимо набрать текст программы на языке Ассемблера в ассемблерной вставке. Формат строчки программы на языке ассемблера в общем виде выглядит следующим образом:

[метка:] мнемоника команды [1ый операнд[,2ой операнд[,3ий операнд]]][[1]](#footnote-1)

1. После набора всей программы необходимо откомпилировать проект известным способом и устранить все ошибки, найденные компилятором в случае, если необходимо.
2. После открытия окна отладчика следует выполнять программу пошагово. Для этого использовать следующие функциональные клавиши:

<F11> – выполнить одну инструкцию c заходом в подпрограмму;

<Shift>+<F11> – выполнение всего кода программы.

1. После выполнения последней команды ассемблерной вставки завершить выполнение программы нажатием клавиши <Shift>+<F5>.

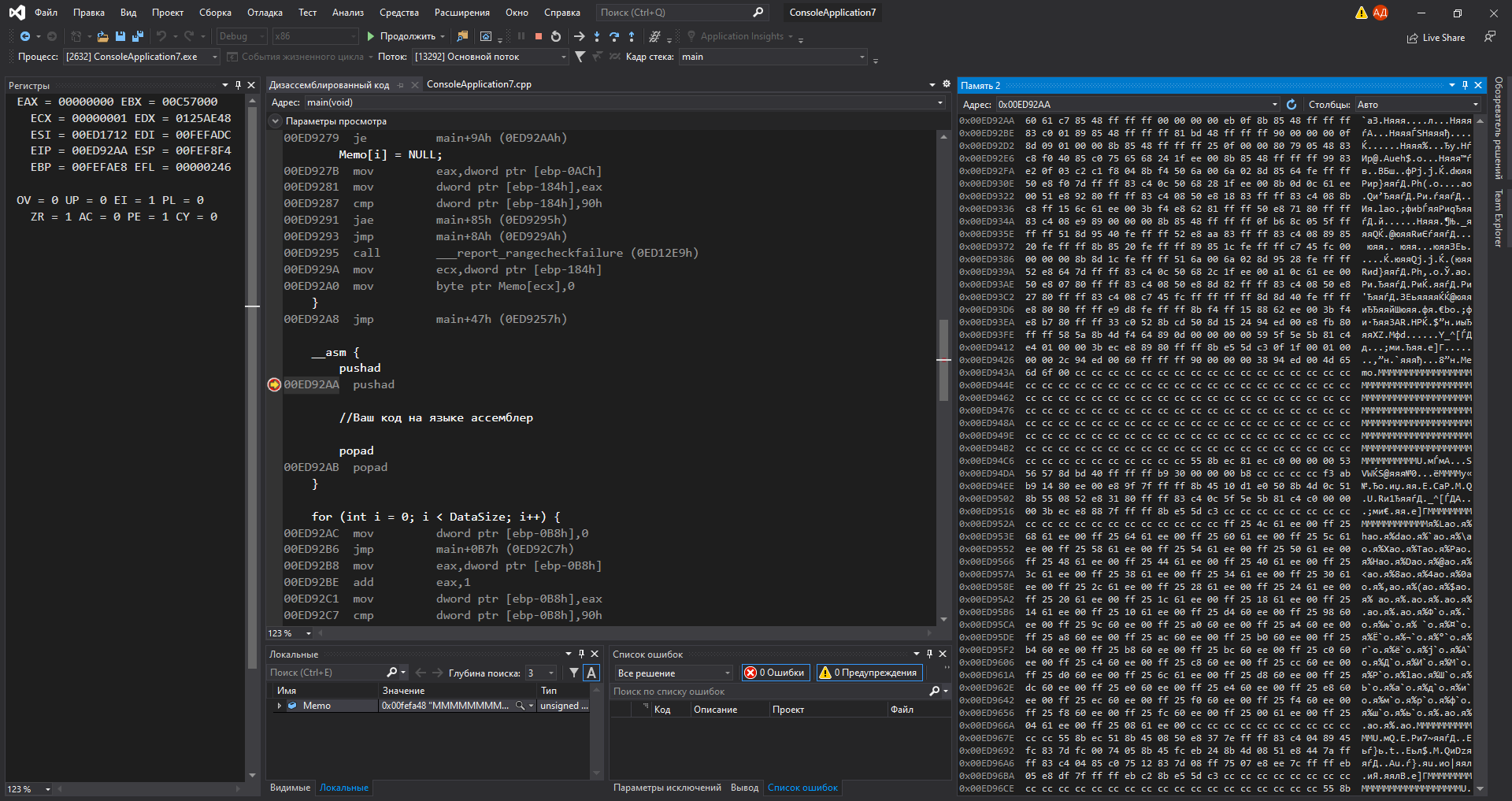


Рисунок 18. Конечный вид IDE для работы с ассемблерными ставками

**Лабораторная работа №1. Регистры, данные и команды пересылки данных Цель работы**

Изучение регистров общего назначения (РОН) и команд пересылки

данных.

**Постановка задачи**

1. Занести число из столбца «Число 1» таблицы 1 и строки, соответствующей заданному варианту, в любой РОН;
2. Занести число из столбца «Число 2» таблицы 1 и строки, соответствующей заданному варианту, в незанятый РОН;
3. Занести число из столбца «Число 3» таблицы 1 и строки, соответствующей заданному варианту, в незанятый РОН;
4. Занести число из столбца «Число 4» таблицы 1 и строки, соответствующей заданному варианту, в незанятый РОН;
5. Занести число из столбца «Число 5» таблицы 1 и строки, соответствующей заданному варианту, в незанятый РОН;
6. Занести число из столбца «Число 6» таблицы 1 и строки, соответствующей заданному варианту, в незанятый РОН;
7. Обменять числа, хранящиеся в РОН после выполнения пунктов 1 и 2, между собой, 4-мя разными способами, не потеряв информацию в занятых РОН;
8. Обменять числа, хранящиеся в РОН после выполнения пунктов 3 и 4, между собой, 4-мя разными способами, не потеряв информацию в занятых РОН;
9. Обменять числа, хранящиеся в РОН после выполнения пунктов 5 и 6, между собой, 4-мя разными способами, не потеряв информацию в занятых РОН;
10. Переслать числа, оказавшиеся в РОН после выполнения пункта с номером, взятым из столбца «Пункт 1» таблицы 1 и строки, соответствующей заданному варианту, в любые 16-разрядные РОН без потери знака;
11. Переслать числа, оказавшиеся в РОН после выполнения пункта с номером, взятым из столбца «Пункт 2» таблицы 1 и строки, соответствующей заданному варианту, в любые 32-разрядные РОН расширенные нулем;

## Примечание: Регистр ESP и EBP использовать НЕЛЬЗЯ !!!

**Содержание отчета**

Отчет по лабораторной работе должен содержать номер, название, цель лабораторной работы, задание на лабораторную работу, текст программы на

языке ассемблера, написанного от руки при подготовке, распечатки формы в активном состоянии с результатами выполнения лабораторной работы в виде дампа памяти и фрагмента программы, содержащей ассемблерную вставку, а также вывод по работе.

**Варианты заданий**

Таблица 1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер вариан  та | Число 1 | Число 2 | Число 3 | Число 4 | Чис ло 5 | Чис ло 6 | Пун кт 1 | Пун кт 2 |
| 1 | 3 050 316 749 | 857 033 805 | 64 001 | 27 946 | 134 | 47 | 8 | 8 |
| 2 | 3 933 384 589 | 971 413 658 | 48 330 | 22 467 | 162 | 5 | 8 | 9 |
| 3 | 3 968 432 562 | 250 761 973 | 45 938 | 25 727 | 210 | 42 | 9 | 8 |
| 4 | 2 669 592 357 | 6 213 853 | 47 036 | 16 800 | 145 | 94 | 9 | 9 |
| 5 | 3 609 857 455 | 2 001 729 633 | 54 633 | 8 498 | 183 | 114 | 8 | 8 |
| 6 | 3 257 266 905 | 1 245 818 205 | 56 002 | 25 418 | 212 | 75 | 8 | 9 |
| 7 | 2 916 235 308 | 987 999 567 | 55 633 | 32 029 | 139 | 26 | 9 | 8 |
| 8 | 2 756 406 257 | 1 506 812 197 | 63 484 | 8 425 | 153 | 4 | 9 | 9 |
| 9 | 4 034 305 306 | 392 802 862 | 35 094 | 11 811 | 137 | 7 | 8 | 8 |
| 10 | 3 159 448 700 | 1 544 211 091 | 57 236 | 22 027 | 243 | 108 | 8 | 9 |
| 11 | 2 488 047 062 | 2 028 847 670 | 47 579 | 22 442 | 250 | 66 | 9 | 8 |
| 12 | 4 247 776 274 | 103 373 407 | 46 130 | 29 961 | 151 | 56 | 9 | 9 |
| 13 | 3 552 292 199 | 1 628 905 995 | 58 404 | 13 389 | 147 | 62 | 8 | 8 |
| 14 | 4 008 698 874 | 306 001 926 | 52 347 | 20 501 | 153 | 74 | 8 | 9 |
| 15 | 2 433 934 443 | 2 015 149 002 | 56 190 | 5 075 | 236 | 69 | 9 | 8 |
| 16 | 3 243 508 463 | 1 098 303 011 | 61 752 | 25 291 | 227 | 60 | 9 | 9 |
| 17 | 3 598 586 003 | 1 700 879 456 | 51 395 | 12 158 | 150 | 72 | 8 | 8 |
| 18 | 3 948 008 498 | 1 823 230 391 | 56 897 | 22 903 | 175 | 89 | 8 | 9 |
| 19 | 4 048 479 298 | 1 670 322 348 | 45 257 | 25 552 | 233 | 110 | 9 | 8 |
| 20 | 2 999 718 865 | 2 089 409 762 | 59 534 | 26 378 | 203 | 59 | 9 | 9 |
| 21 | 2 711 728 559 | 1 472 656 564 | 62 574 | 32 346 | 204 | 51 | 8 | 8 |
| 22 | 3 683 606 123 | 335 322 861 | 64 260 | 6 699 | 224 | 112 | 8 | 9 |
| 23 | 3 645 284 490 | 1 559 350 478 | 59 482 | 3 327 | 149 | 78 | 9 | 8 |
| 24 | 3 821 456 732 | 183 754 094 | 42 930 | 5 063 | 223 | 66 | 9 | 9 |
| 25 | 2 516 094 474 | 1 472 212 419 | 51 469 | 16 564 | 145 | 86 | 8 | 8 |
| 26 | 4 194 231 035 | 954 481 758 | 33 811 | 5 512 | 171 | 88 | 8 | 9 |
| 27 | 2 398 029 387 | 3 261 100 | 63 166 | 13 971 | 224 | 97 | 9 | 8 |
| 28 | 3 401 178 541 | 2 021 332 153 | 38 118 | 32 100 | 184 | 55 | 9 | 9 |
| 29 | 3 312 549 825 | 653 774 000 | 45 597 | 12 799 | 246 | 77 | 8 | 8 |
| 30 | 4 130 192 976 | 1 564 933 669 | 55 114 | 16 506 | 175 | 83 | 8 | 9 |
| 31 | 4 138 113 708 | 1 815 058 639 | 42 042 | 5 573 | 219 | 58 | 9 | 8 |
| 32 | 4 286 093 495 | 1 084 073 624 | 55 016 | 31 876 | 227 | 110 | 9 | 9 |
| 33 | 3 121 711 706 | 1 966 382 217 | 38 040 | 1 449 | 242 | 90 | 8 | 8 |
| 34 | 2 374 644 833 | 962 127 780 | 54 116 | 25 875 | 199 | 88 | 8 | 9 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Номер вариан  та | Число 1 | Число 2 | Число 3 | Число 4 | Чис ло 5 | Чис ло 6 | Пун кт 1 | Пун кт 2 |
| 35 | 2 809 635 978 | 1 981 105 341 | 34 106 | 5 212 | 189 | 61 | 9 | 8 |
| 36 | 4 138 500 438 | 1 767 390 115 | 39 427 | 21 800 | 211 | 18 | 9 | 9 |
| 37 | 2 156 758 467 | 660 985 841 | 35 423 | 28 994 | 176 | 22 | 8 | 8 |
| 38 | 4 217 006 650 | 1 168 671 945 | 62 887 | 6 141 | 164 | 80 | 8 | 9 |
| 39 | 4 047 827 015 | 1 821 383 074 | 58 379 | 22 630 | 191 | 96 | 9 | 8 |
| 40 | 3 559 739 825 | 256 920 863 | 33 198 | 6 043 | 165 | 66 | 9 | 9 |
| 41 | 3 780 056 098 | 1 603 982 851 | 51 278 | 17 711 | 183 | 62 | 8 | 8 |
| 42 | 3 797 296 420 | 1 908 629 559 | 48 250 | 21 347 | 240 | 120 | 8 | 9 |
| 43 | 3 736 924 978 | 1 296 290 749 | 32 944 | 32 763 | 157 | 44 | 9 | 8 |
| 44 | 2 543 238 445 | 372 228 216 | 44 181 | 4 117 | 131 | 106 | 9 | 9 |
| 45 | 2 786 920 617 | 47 245 825 | 35 912 | 17 475 | 240 | 82 | 8 | 8 |
| 46 | 2 570 192 319 | 632 432 585 | 41 490 | 8 458 | 202 | 120 | 8 | 9 |
| 47 | 3 536 602 848 | 621 455 134 | 33 441 | 28 950 | 193 | 116 | 9 | 8 |
| 48 | 2 518 445 482 | 93 653 929 | 50 079 | 28 225 | 153 | 38 | 9 | 9 |
| 49 | 3 540 914 521 | 1 864 588 387 | 43 010 | 15 728 | 200 | 28 | 8 | 8 |
| 50 | 3 553 032 756 | 2 081 070 464 | 40 595 | 567 | 161 | 16 | 8 | 9 |
| 51 | 3 234 044 570 | 1 430 006 929 | 49 279 | 11 831 | 209 | 34 | 9 | 8 |
| 52 | 3 161 648 983 | 1 268 418 336 | 60 595 | 2 650 | 149 | 27 | 9 | 9 |
| 53 | 3 322 204 893 | 256 698 229 | 49 297 | 8 110 | 253 | 5 | 8 | 8 |
| 54 | 4 161 566 815 | 1 396 434 740 | 37 590 | 17 767 | 241 | 49 | 8 | 9 |
| 55 | 4 219 198 386 | 769 555 691 | 45 524 | 28 107 | 183 | 100 | 9 | 8 |
| 56 | 3 057 003 449 | 2 122 839 732 | 62 337 | 5 848 | 167 | 86 | 9 | 9 |
| 57 | 3 222 886 638 | 50 497 194 | 55 352 | 26 187 | 141 | 28 | 8 | 8 |
| 58 | 4 075 678 955 | 1 348 534 018 | 37 269 | 1 168 | 236 | 70 | 8 | 9 |
| 59 | 2 417 251 151 | 1 059 178 430 | 65 004 | 29 622 | 241 | 102 | 9 | 8 |
| 60 | 2 782 443 731 | 1 904 828 683 | 65 166 | 20 558 | 228 | 37 | 9 | 9 |

**Краткие теоретические сведения**

Микропроцессор Intel семейства P6 содержит восемь регистров общего назначения (РОН), которые используются для хранения операндов и результатов команд, а также для формирования адреса при обращении к данным в памяти.

РОН микропроцессора Intel P6, их разрядность и вложенность приведены в таблице 2.

Таблица 2.

31 16 15 8 7 0 ← номера битов

|  |  |  |
| --- | --- | --- |
|  | AX | |
| AH | AL |
|  | BX | |
| BH | BL |
|  | CX | |
| CH | CL |
|  | DX | |
| DH | DL |
|  | BP | |
|  | SI | |
|  | DI | |
|  | SP | |

EAX EBX ECX EDX EBP ESI EDI

ESP

При обращении к ячейкам памяти, адрес ячейки памяти может указываться косвенно, т.е. адрес может быть указан в каком-либо РОН или с помощью математического выражения следующего вида:

пппппппп

Первое слагаемое называется базой, второе – индекс, умноженный на масштаб, третье – константа. Формула может состоять из любого сочетания слагаемых в квадратных скобках (напомним, что в квадратных скобках здесь указаны слагаемые, которые могут полностью отсутствовать) и любого элемента из каждой строчки в круглых скобках. Следует напомнить, что операнд, задающий адрес ячейки памяти, указывается в квадратных скобках.

Все команды на языке ассемблера записываются в виде мнемоники команды и разного количества операндов. Мнемоника — это сокращение английских слов, выражающих смысл команды.

К командам пересылки данных относятся:

MOV - от англ. Move – движение, т.е. перемещение информации; PUSH - от англ. Push – поместить, т.е. занести информацию;

POP - от англ. POP - выталкивание, т.е. забрать информацию; XCHG - от англ. Exchange – обмен информацией;

BSWAP - от англ. Byte Swap – байтовый обмен;

MOVSX - от англ. Move with Sign Extended – перемещение информации со знаковым расширением;

MOVZX - от англ. Move with Zero Extended – перемещение информации с расширением нулем;

LEA - от англ. Load Effective Address – загрузить эффективный адрес.

**Команда MOV**

Формат команды: MOV П,И

где П – приемник информации. В качестве приемника может использоваться РОН или ячейка памяти;

И – источник информации. В качестве источника информации могут использоваться РОН, ячейка памяти или константа (непосредственные данные, указанные в самой команде ассемблера).

Команда MOV копирует данные из источника в приемник. Приемник и источник должны быть одинаковой размерности! Нельзя в одной команде MOV пересылать данные между двумя ячейками памяти! При записи в память, т.е., если в качестве приемника указан адрес ячейки памяти, информация размером в Byte (8 бит) будет записана непосредственно в ячейку памяти с адресом, указанным в команде. При записи информации размером в Word (2 байта или 16 бит), будут задействованы две соседние ячейки памяти, начиная с адреса, указанного в команде, причем первым будет записан младший байт информации, а в соседнюю ячейку (в сторону увеличения адресов), старший байт информации. При записи информации размером Integer (2 слова, 4 байта,32 бита), будут задействованы четыре соседних ячейки памяти начиная с адреса, указанного в команде, причем, первым будет записан младший байт младшего слова информации, а в соседние ячейки (в сторону увеличения адресов), соответственно, старший байт младшего слова, младший байт старшего слова и старший байт старшего слова (младшим байтом или словом называют байт или слово с меньшими номерами битов, например, в слове младшим байтом называют байт, который состоит из битов с нулевого по седьмой, а старшим тот, который состоит из битов с восьмого по пятнадцатый).

Примеры:

MOV AL,10 Занести число 10 в регистр AL. После выполнения

команды AL=0A.

MOV BX,$FA Занести число FA16 (25010) в регистр BX. В данной

строке примера использован символ «$» перед числом, который является обязательным префиксом для шестнадцатеричного числа. После выполнения команды BX=00FA. Несмотря на то, что в команде указана байтовая константа, компилятор оценит приемник, в качестве которого здесь выступает 16-ти разрядный регистр, и автоматически расширит константу незначащими нулями слева.

MOV ECX,EDX Занести значение регистра EDX в регистр ECX.

Совершенно очевидно, что EDX сохранит свое значение.

MOV [EBX],BH Занести в ячейку памяти (т.к. приемник указан в

квадратных скобках) с адресом, хранящимся в регистре EBX, один байт (так как источником является 8-ми разрядный регистр) со значением, хранящимся в регистре BH.

MOV Byte Ptr [ESI], $F0 Занести в ячейку памяти (т.к. приемник указан в

квадратных скобках) с адресом, хранящимся в регистре ESI, один байт со значением F016 (24010). В данном случае определить размер источника без дополнительных указаний не представляется возможным, так как это может быть и F016, и 00F016, и 000000F016 и любая из этих констант может быть записана в память, заняв, соответственно, одну, две или четыре соседних ячейки памяти. В таких случаях размер данных указывается специальным префиксом, в данном случае Byte Ptr, который четко указывает, что заноситься будет один байт.

MOV Word Ptr [ECX+EDX\*2], $1234 Занести в ячейку памяти с адресом,

вычисленным по выражению ECX+EDX\*2, одно слово со значением 123416 (466010). При этом по адресу ECX+EDX\*2 будет занесен байт $34, а по адресу ECX+EDX\*2+1 байт $12.

MOV DWord Ptr [ECX+EBX+$A], $12345678 Занести в ячейку памяти с

адресом, вычисленным по выражению ECX+EBX+$A, двойное слово со значением 1234567816 (1908874410). При этом по адресу ECX+EBX+$A будет занесен байт $78, по адресу ECX+EBX+$A+1 байт $56, по адресу ECX+EBX+$A+2 байт $34, а по адресу ECX+EBX+$A+3 байт $12.

**Команда PUSH**

Формат команды: PUSH И

где И – источник информации. В качестве источника информации могут использоваться РОН, ячейка памяти или константа (непосредственные данные, указанные в самой команде ассемблера).

Команда PUSH копирует данные из источника в вершину стека. В качестве источника могут использоваться только 16-ти и 32-ух разрядные данные! Стеком называется область памяти в общем адресном пространстве, которая работает по принципу FILO (First In Last Out – первым вошел последним вышел). Для работы со стеком используются специальные команды, а также стек используется автоматически при вызове подпрограмм.

В самом простом случае стек используется для быстрого временного сохранения и восстановления значений РОН. Языки высокого уровня используют стек для передачи параметров в процедуры и функции.

**Внимание: В процессорах Intel P6 адрес вершины стека после занесения данных уменьшается.**

Примеры:

PUSH AX Поместить в стек значение регистра AX.

PUSH Word Ptr $FA Поместить в стек значение слова FA16 (25010).

Таким образом в стек запишется 00FA16.

PUSH DWord Ptr [EBX] Поместить в стек длинное слово из четырех

соседних ячеек, первая из которых имеет адрес с номером, хранящимся в EBX.

Команда PUSH имеет модификации:

PUSHA – занести в стек все 16-ти разрядные РОН в следующем порядке AX,CX,DX,BX,SP,BP,SI,DI;

PUSHAD – занести в стек все 32-ух разрядные РОН в следующем порядке EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI;

**Команда POP**

Формат команды: POP П

где П – приемник информации. В качестве приемника информации могут использоваться РОН или ячейка памяти.

Команда POP копирует данные из вершины стека в приемник.

**Внимание:**

1. В качестве приемника могут использоваться только 16-ти и 32-ух разрядные хранилища!
2. Количество команд PUSH должно соответствовать количеству команд POP.

Примеры:

POP AX Восстановить значение регистра AX из стека.

POP DWord Ptr [EBX] Восстановить содержимое четырех соседних ячеек,

первая из которых имеет адрес с номером, хранящимся в EBX, из стека.

Команда POP имеет модификации:

POPA – восстановить значения всех 16-ти разрядных РОН из стека; POPAD – восстановить значения всех 32-ух разрядных РОН из стека;

**Команда XCHG**

Формат команды: XCHG П,И

где П – приемник информации. В качестве приемника может использоваться РОН;

И – источник информации. В качестве источника информации могут использоваться РОН или ячейка памяти.

Команда XCHG обменивает данные между источником и приемником. **Внимание:**

1. Приемник и источник должны быть одинаковой размерности!
2. Нельзя в команде XCHG обменивать данные между двумя ячейками памяти!

**Команда BSWAP**

Формат команды: BSWAP П

где П – приемник информации. В качестве приемника может использоваться только 32-ух разрядный РОН.

Команда BSWAP обменивает в двойном слове между собой значения младшего и старшего байтов, а также средних байтов. Схема обмена представлена на рисунке 11.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  |  |  |  |
| --- | --- | --- | --- |
| Byte 3 | Byte 2 | Byte 1 | Byte 0 |



Рисунок 11. Схема обмена байтами в команде BSWAP.

**Команда MOVZX**

Формат команды: MOVZX П,И

где П – приемник информации. В качестве приемника может использоваться 16-ти или 32-ух разрядный РОН;

И – источник информации. В качестве источника информации могут использоваться 8-ми или 16-ти разрядные РОН или ячейки памяти.

Команда MOVZX перемещает данные из источника в приемник, увеличивая размерность данных в соответствии с размером приемника, путем добавления нулевых битов слева, как это представлено на рисунке 12.

0 0 0 0 0 0 0 0 8-ми разрядный источник

7 Byte 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

16-ти разрядный приемник

8-ми разрядный источник

7 Byte 0 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0

32-ух разрядный приемник

16-ти разрядный источник

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | Byte 3 | 24 | 23 | Byte 2 | 16 | 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

32-ух разрядный приемник

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | Byte 3 | 24 | 23 | Byte 2 | 16 | 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

Рисунок 12. Схема увеличения размерности данных в команде MOVZX.

**Команда MOVSX**

Формат команды: MOVSX П,И

где П – приемник информации. В качестве приемника может использоваться 16-ти или 32-ух разрядный РОН;

И – источник информации. В качестве источника информации могут использоваться 8-ми или 16-ти разрядные РОН или ячейки памяти.

Команда MOVSX перемещает данные из источника в приемник, увеличивая размерность данных в соответствии с размером приемника, путем добавления значения старшего бита источника в расширяемые биты приемника, как это представлено на рисунке 13.

8-ми разрядный источник



7

Byte 0

0

16-ти разрядный приемник

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

8-ми разрядный источник



7

Byte 0

0

… … ...

… … ...

… … ...

32-ух разрядный приемник

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | Byte 3 | 24 | 23 | Byte 2 | 16 | 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

16-ти разрядный источник

… … ...

… … ...

0

Byte 0

7

8

Byte 1

15

32-ух разрядный приемник

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | Byte 3 | 24 | 23 | Byte 2 | 16 | 15 | Byte 1 | 8 | 7 | Byte 0 | 0 |

Рисунок 13. Схема увеличения размерности данных в команде MOVSX.

**Команда LEA**

Формат команды: LEA П,И

где П – приемник информации. В качестве приемника может использоваться только 16-ти или 32-ух разрядные РОН. (В операционной системе Windows необходимо использовать 32-ух разрядные РОН);

И – источник информации. В качестве источника информации могут использоваться 16-ти или 32-ух разрядные ячейки памяти.

Команда LEA вычисляет адрес операнда-источника и помещает его в приемник. Вычисление производиться самим процессором во время выполнения программы.

Пример:

1. var
2. Data : Array[1..100] of Byte;
3. begin
4. asm
5. PUSHAD
6. LEA EBX,Data
7. LEA ECX,Data+5
8. MOV Byte Ptr [EBX],$55
9. MOV Word Ptr [ECX],$1234
10. POPAD
11. end;
12. end;

Программа демонстрирует пример совмещения кода на языке высокого уровня и ассемблера:

* 1. Первые две строчки программы описывают переменную *Data* типа массив на сто байтовых элементов.
  2. Третья строка начинает тело программы, а двенадцатая закрывает.
  3. Четвертая строка открывает ассемблерную вставку, а одиннадцатая закрывает. Для нормального функционирования программы, которую сформирует компилятор, необходимо запомнить значения всех регистров перед началом выполнения ассемблерной вставки (строка 5), а по ее завершении восстановить (строка 10). Дело в том, что компилятор строго контролирует содержимое регистров, используя их для выполнения программы. Ассемблерная вставка – это код, сразу предназначенный для процессора, и, что в нем будет запрограммировано, компилятор предугадать не может, и, соответственно, не может освободить нужные регистры. Ассемблерная же вставка может изменить значения регистров, а значит, дальнейшее нормальное выполнение программы может быть нарушено. Для этого значения регистров временно сохраняют в стеке, а потом восстанавливают.
  4. Шестая строка вычисляет адрес первого байта массива *Data* и сохраняет его в регистре EBX. Дело в том, что операционная система *Windows* может загрузить программу по любому адресу общей памяти. Все будет зависеть от уже загруженных программ, динамических библиотек, состояния системы и т.д. Чтобы использовать массив *Data*, если неизвестно, по какому адресу загружена программа и где выделено место для массива *Data*, нужно в программе вычислить его адрес, для чего и применяется команда LEA.
  5. Седьмая строка вычисляет адрес 6-ого байта массива *Data* (у первого байта адрес метки *Data*, у второго *Data+1*, у третьего *Data+2*, и т.д.) и сохраняет его в регистре ECX. Смысл 8-ой и 9-ой строк понятен (см. команду MOV). В результате выполнения программы получим:

Data[1] = 5516 (8510);

Data[6] = 3416 (5210);

Data[7] = 1216 (1810);

**Примерные контрольные вопросы и задачи**

* 1. Чему будут равны регистры CH, CL, CX после выполнения команды MOV ECX,876543210 ?
  2. Чему будут равны регистры AX, BX и EDX после выполнения следующего фрагмента программы:

MOV CL,$127 MOV CH,$128 MOVSX AX,CH MOVZX BX,CL MOVSX EDX,BX ?

* 1. Чему будет равен регистр EDX после выполнения следующего фрагмента программы:

LEA EBX,Data

MOV ECX,$78FA564D MOV EDX,$12E90BC3 MOV [EBX+3],EDX

MOV [EBX+5],ECX

MOV Word Ptr [EBX+2],$B3 MOV EDX,[EBX+3] ?

* 1. Что такое дамп памяти?
  2. Что такое стек?
  3. Назовите все восьмиразрядные РОН?
  4. Назовите все шестнадцатиразрядные РОН?
  5. Назовите все тридцатидвухразрядные РОН?
  6. Чему будут равны регистры BL, BX, BH после выполнения следующего фрагмента программы:

MOV EBX,1988776655 BSWAP EBX ?

* 1. Что означает префикс DWord Ptr и где он используется?
  2. Чему будут равны регистры BX и CX после выполнения следующего фрагмента программы:

MOV EAX,$12345678 PUSH EAX

POP BX POP CX ?

* 1. Чему равен старший бит числа 200, если его хранить в памяти как слово (Word)?

# Лабораторная работа №2. Арифметические команды

**Цель работы**

Изучение арифметических команд и получение навыка работы с ними.

**Постановка задачи**

Занести числа в память:

a= 5'678'901'234;

b= 6'789'012'345;

c= 7'890'123'456;

d= 8'901'234'567;

e= 9'012'345'678; f=10'123'456'789.

Произвести операции над числами a,b,c,d,e,f по вариантам, заданным в

таблице 3. Результат вывести в память и распечатать в *Memo*.

**Содержание отчета**

Отчет по лабораторной работе должен содержать номер, название, цель лабораторной работы, задание на лабораторную работу, текст программы на языке ассемблера, написанного от руки при подготовке, распечатки формы в активном состоянии с результатами выполнения лабораторной работы в виде дампа памяти и фрагмента программы, содержащей ассемблерную вставку, а также контрольный пример и выводы по работе.

**Варианты заданий**

Таблица 3.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***Номер варианта*** | ***Выражение*** | | ***Номер варианта*** | ***Выражение*** | | ***Номер варианта*** | ***Выражение*** |
| **1** | 𝟐 ∗ 𝒄 − 𝒅 + 𝟐𝟑  𝒂⁄𝟒 − 𝟏 | | **17** | 𝟐 ∗ 𝒄 + 𝒅 ∗ 𝟓𝟏  𝒅 − 𝒂 − 𝟏 | | **33** | *c*  33  *b*  4  *ac*  1  *b* |
| **2** |  | 𝒄 + 𝟒 ∗ 𝒅 − 𝟏𝟐𝟑 | **18** | 𝟐 ∗ 𝒄 + 𝒅 + 𝟐𝟑  𝟒  𝒅 − 𝒂 − 𝟏 | | **34** | 2*b*  38*c*  *b*  *a*  1  *c* |
| 𝟏 − 𝒂⁄𝟐 |
| **3** | −𝟐 ∗ 𝒄 + 𝒅 ∗ 𝟖𝟐  𝒂⁄𝟒 − 𝟏 | | **19** | 15*a*  *b*  *a*  4  *ab* 1 | | **35** | *c*  28*d*  4  *a*  *c*  1  *d* |
| **4** | 𝟐 ∗ 𝒄 + 𝒅 − 𝟓𝟐  𝒂⁄𝟒 + 𝟏 | | **20** |  25  *c*  *ba*  *a* | | **36** | *ab* 1  4  41  *ba*  *c* |
| 1  *cb*  2 |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | 𝒄⁄𝟒 − 𝒅 ∗ 𝟔𝟐  𝒂 ∗ 𝒂 + 𝟏 | | **21** | 4*a* 1  *b*  2  *bc*  5 | **37** | 1  6*a*  *b*  2  *c*  *a*  *d* |
| **6** |  | −𝟐 ∗ 𝒄 − 𝒅 + 𝟓𝟑 | **22** | 8*b* 1 *c*  *a*  *bc*  2 | **38** | *ab*  2*c*  41  *b*  1  *c* |
| 𝒂⁄𝟒 − 𝟏 |
| **7** | 𝒄⁄𝟒 − 𝒅 ∗ 𝟔𝟐  𝒂 ∗ 𝒂 + 𝟏 | | **23** | 4*a*  *b*  1  *c*  *a b* | **39** | 4*b*  *ca*  *b*  *c*  1 28 |
| **8** | 𝟐 + 𝒄 − 𝒅 ∗ 𝟐𝟑  𝟐 ∗ 𝒂𝟐 − 𝟏 | | **24** | 4*b* 1  *c*  12*c*  *a*  *b* | **40** | 2*c*  *a*  21  *c*  1  *ab* |
| **9** | 𝟐 ∗ 𝒄 − 𝒅/𝟑  𝒃 − 𝒂/𝟒 | | **25** | *b*  *cb*  *a*  4  *ab* 1 | **41** | 4  3*a c*  *c*  *b* 1  *a* |
| **10** | 𝟒 ∗ 𝒄 + 𝒅 − 𝟏  𝒄 − 𝒂/𝟐 | | **26** | *a*  *c*  28  *b*  4*ba*  1 | **42** | 8*b*  1  *c*  2*a*  *b*  *c* |
| **11** | 𝟐 ∗ 𝒄 − 𝒅 ∗ 𝟒𝟐  𝒄 + 𝒂 − 𝟏 | | **27** | *c*  24  *a*  *b*  2*ac*  1 | **43** | 4*a*  1  *b*  *cb* 18  *a* |
| **12** | 𝟐𝟓 − 𝒅 + 𝟐  𝒄  𝒃 + 𝒂𝟐 − 𝟏 | | **28** | 2*b*  *a*  *bc*  *c*  1  4 | **44** | 4*b*  1  *c*  2*c*  *ac*  *b* |
| **13** | 𝒄 − 𝒅 + 𝟑𝟑  𝟐  𝟐 ∗ 𝒂𝟐 − 𝟏 | | **29** | 41  *d*  1  4  *c*  *ad b* | **45** | *b*  *c*  *a*  *b* 4  *ab* 1 |
| **14** | 𝟒 ∗ 𝒄 − 𝒅 + 𝟐𝟑  𝟐  𝒂𝟐 − 𝟏 | | **30** | *a*  4*b*  1  *c*  *ab*  31 | **46** | *a*  *cb*  28  4*b*  1  *a* |
| **15** | 𝒄 ∗ 𝒅 + 𝟐𝟑  𝒂  𝟐 − 𝟒 ∗ 𝒅 − 𝟏 | | **31** | *b*  4*c*  *a*  *c*  *b*  1 | **47** | *cb*  24  *a*  *b*  1 2*c* |
| **16** | 𝒄 + 𝟑 ∗ 𝒂/𝟐  𝒅  𝒄 − 𝒂 + 𝟏 | | **32** | 21  *ac*  4  1  *c*  *b a* | **48** | 2*b*  *a*  *b*  *c*  *c*  1  4 |

**Краткие теоретические сведения**

**Регистр флагов**

При выполнении некоторых команд процессору необходима дополнительная информация, так же как и после выполнения некоторых

команд необходимо сохранить дополнительную информацию о результатах выполнения команды. Чаще всего необходимость в дополнительной информации при выполнении команд возникает при выполнении арифметических команд. Поскольку процессор может оперировать данными определенных размеров (Byte, Word, LongWord), а при выполнении арифметических операций результат может не уместиться в используемый тип данных, то необходимо эту дополнительную информацию сохранять для дальнейшего учета. Местом хранения этой дополнительной информации является регистр флагов EFLAGS. Регистр флагов представляет собой 32- битный регистр специального назначения (РСН), где фиксируется информация о текущем состоянии процессора. Многие их его битов используются при выполнении команд и устанавливаются после выполнения команд. Расположение битов в регистре флагов представлено в таблице 4. Первая строка таблицы содержит номера битов регистра флагов, вторая – их аббревиатуры.

Таблица 4.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ID | VIP | VIF | AC | VM | RF | 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

Флаг **CF** называется флагом переноса (от английского *Carry Flag*). Флаг переноса устанавливается (становиться равным 1) при переносе (заёме), возникающем при сложении (вычитании). Иначе сбрасывается (становиться равным 0). Может быть установлен и в других случаях в зависимости от выполняемой команды.

Флаг **ZF** – флаг нулевого результата (*Zero Flag*). Если результат последней операции равен нулю, то флаг устанавливается, иначе сбрасывается.

Флаг **AF** – флаг вспомогательного переноса (*Auxiliary Carry Flag*). Устанавливается при переносе (заёме) из бита 3 (в бит 3) при сложении (вычитании). Используется только для двоично-десятичной арифметики, которая оперирует исключительно младшими байтами.

Флаг **SF** – флаг знака (*Sign Flag*). Дублирует старший бит результата, который при использовании дополнительного кода соответствует знаку числа.

Флаг **OF** – флаг переполнения (*Overflow Flag*). Сигнализирует о потере старшего бита результата в связи с переполнением разрядной сетки при работе со знаковыми числами. Равен 1, когда:

* при сложении был перенос из старшего бита, но не было переноса в старший бит;
* при сложении был перенос в старший бит, но не было переноса из старшего бита;
* при вычитании был заем в старший бит, но не было заёма из старшего

бита; бит.

- при вычитании был заем из старшего бита, но не было заёма в старший

Флаг **PF** – флаг четности (*Parity Flag*). Устанавливается в 1 при наличие четного числа единиц в двоичном представлении младшего байта результата (**ЧЕТНОСТЬ НЕ ПО МАТЕМАТИКЕ**!)

Флаг **DF** – флаг направления (*Direction Flag*). Используется в командах работы со строками (цепочками). При DF=1 регистр(ы) индекса, используемый(ые) в командах работы со строками, увеличивается(ются) на 1 при каждом следующем выполнении команды, при DF=0 - регистр(ы) индекса уменьшается(ются) на 1.

Остальные флаги являются достаточно специфичными и выходят за рамки данного пособия. Их описание можно найти, например, в [2-3].

В Visual Studio обозначения флагов отличается от классического их обозначения. Ниже приведена таблица соответствий названий.

|  |  |  |
| --- | --- | --- |
| Названия флагов | Флаги в Visual Studio | Флаги регистров |
| Флаг переноса | CY | CF |
| Флаг нулевого результата | ZR | ZF |
| флаг вспомогательного переноса | AC | AF |
| флаг знака | PL | SF |
| флаг переполнения | OV | OF |
| флаг четности | PE | PF |
| флаг направления | UP | DF |

К арифметическим командам относятся:

ADD (Add) – Сложить;

ADC (Add with Carry) – Сложить с учетом переноса; XADD (eXchange and Add) – Обменять и сложить;

SUB (Subtract) – Вычесть;

SBB (Subtract with borrow) – Вычесть с учетом заёма; INC (Increment) – Приращение;

DEC (Decrement) – Декремент (отрицательное приращение),

уменьшение;

NEG (Negative) – Выдать число с противоположным

(негативным) знаком; MUL (Multiplication) – Умножение;

IMUL (Integer Multiplication) – Умножение с учета знака; DIV (Division) – Деление;

IDIV (Integer Division) – Деление с учетом знака;

CWD (Convert Word to Double Word) – Преобразовать слово в двойное

слово;

CDQ (Convert Double Word to Quad Word) – Преобразовать двойное

слово в четверное слово; CBW (Convert Byte to Word) – Преобразовать байт в слово;

CWDE (Convert Word to Double Word Extended) – Преобразовать слово в

двойное слово, дополнительная функция.

**Команды ADD, ADC, XADD**

Формат команды: ADD П,И

ADC П,И XADD П,И

где П – приемник информации. В качестве приемника может использоваться РОН или ячейка памяти;

И – источник информации. В качестве источника информации могут использоваться РОН, ячейка памяти или константа (непосредственные данные, указанные в самой команде ассемблера). Приемник и источник должны быть одинаковой размерности! Нельзя в одной команде в качестве и приемника и источника использовать ячейки памяти!

Команда **ADD** выполняет следующую операцию: П=П+И; Команда **ADC** выполняет следующую операцию:

П=П+И+значение флага CF; Команда **XADD** выполняет следующие операции: 1) П↔И, 2) П=П+И;

Все три команды влияют на флаги OF,SF,ZF,AF,PF,CF, т.е. эти флаги принимают значения по результатам выполнения команды.

Связка команд ADD и ADC, как правило, используются для сложения чисел повышенной точности. Для примера, сложим два 64-битных числа. Поскольку в архитектуре IA32, к которой относится большинство микропроцессоров ПК, нет 64-х разрядных РОН, то хранение и сложение чисел большей разрядности, чем 32, приходиться выполнять по частям, задействовав по два 32-х разрядных РОН для каждого числа. Пусть одно число находиться в паре регистров EDX:EAX (EAX – младшее двойное слово, EDX – старшее двойное слово), а другое - в паре регистров EBX:ECX. При разбиении числа на части необходимо максимально использовать разряды РОН для младшей части. Рассмотрим, для примера, число 10'000'000'00010. В шестнадцатеричной системе счисления это число равно 2'540B'E40016. Как видно, это число занимает больше 32-х разрядов. При разбиении на части, младшая часть будет равна 540B'E40016, а старшая 0000'000216. Программа сложения 64-битных чисел будет выглядеть так:

ADD EAX,ECX ADC EDX,EBX

Если при сложении младших двойных слов произошел перенос из старшего (15-ого) разряда, т.е. флаг CF стал равным 1, то он будет учтен в команде ADC при сложении старших частей. Стоит отметить, что сложение обязательно начинать с младших частей.

В большинстве встраиваемых систем используются 8-ми и 16-ти разрядные микропроцессоры и микроконтроллеры с соответствующей им разрядностью РОН. Необходимость сложения чисел по частям для таких систем возникает уже для чисел больше 25510=FF16, т.е. для чисел, занимающих более 8-ми двоичных разрядов. Часто для 8-ми разрядных систем возникает ситуация, что результирующие числа могут не умещаться и в два байта. При этом сложение чисел, занимающих несколько байтов в 8-ми разрядной системе, ведется последовательно для всех частей, продвигаясь от младшей части к старшей, причем для младших частей применяется команда ADD, а для всех остальных ADC. Между командами ADC нельзя применять команды, изменяющие значение флага CF.

**Команды SUB и SBB**

Формат команды: SUB П,И

SBB П,И

где П – приемник информации. В качестве приемника может использоваться РОН или ячейка памяти;

И – источник информации. В качестве источника информации могут использоваться РОН, ячейка памяти или константа (непосредственные данные, указанные в самой команде ассемблера).

Приемник и источник должны быть одинаковой размерности! Нельзя в одной команде в качестве и приемника и источника использовать ячейки памяти!

Команда **SUB** выполняет следующую операцию: П=П-И; Команда **SBB** выполняет следующую операцию:

П=П-И-значение флага CF;

Команды SUB и SBB влияют на флаги OF,SF,ZF,AF,PF,CF, т.е. эти флаги принимают значения по результатам выполнения команды.

Связка команд SUB, SBB, как и ADD, ADC, как правило, используются для вычитания чисел повышенной точности, когда младшие части чисел вычитаются командой SUB, а остальные командой SBB.

**Команда INC, DEC**

Формат команды: INC П

DEC П

где П – приемник информации. В качестве приемника может использоваться РОН или ячейка памяти.

Команда **INC** выполняет следующую операцию: П=П+1; Команда **DEC** выполняет следующую операцию: П=П-1;

Команда INC увеличивает содержимое приемника на 1, а команда DEC – уменьшает на 1. Отличие от команд сложения и вычитания в том, что не затрагивается флаг CF (флаг переноса). Остальные влаги OF,SF,ZF,AF,PF устанавливаются в соответствии с результатом выполнения команды.

**Команда NEG**

Формат команды: NEG П

где П – приемник информации. В качестве приемника информации могут использоваться РОН или ячейка памяти.

Команда NEG выполняет следующую операцию: П=0-П;

Команда NEG изменяет значение знака содержимого приемника, что вытекает из математического смысла вычитания из нуля. Команда NEG влияет на флаги OF,SF,ZF,AF,PF,CF, т.е. эти флаги принимают значения по результатам выполнения команды вычитания из нуля приемника.

**Команда MUL**

Формат команды: MUL И

где И – источник информации. В качестве источника информации могут использоваться РОН или ячейка памяти.

Команда MUL выполняет беззнаковое умножение. При выполнении команды используются неявные операнды:

* если И 8-ми разрядный, то AX=AL\*И. Если в результате AH=0, то CF=OF=0, иначе CF=OF=1;
* если И 16-ти разрядный, то DX:AX=AX\*И. Если в результате DX=0, то CF=OF=0, иначе CF=OF=1;
* если И 32-ух разрядный, то EDX:EAX=EAX\*И. Если в результате EDX=0, то CF=OF=0, иначе CF=OF=1.

Значение флагов SF,ZF,AF,PF неопределенно, т.е. стоит считать их значения после выполнение команды случайными.

**Команда IMUL**

Команда IMUL выполняет операцию умножения с учетом знака.

1. ый формат команды (однооперандный): IMUL И

где И – источник информации. В качестве источника информации могут использоваться РОН или ячейка памяти.

В данном формате команды используются неявные операнды:

* + если И 8-ми разрядный, то AX=AL\*И. Если в результате AH=0 или AH=FF16, то CF=OF=0, иначе CF=OF=1;
  + если И 16-ти разрядный, то DX:AX=AX\*И. Если в результате DX=0 или DX=FFFF16, то CF=OF=0, иначе CF=OF=1;
  + если И 32-ух разрядный, то EDX:EAX=EAX\*И. Если в результате EDX=0 или EDX=FFFF'FFFF16, то CF=OF=0, иначе CF=OF=1.

Значение флагов SF,ZF,AF,PF неопределенно, т.е. стоит считать их значения после выполнение команды случайными.

1. ой формат команды (двухоперандный): IMUL П,И

где И – источник информации. В качестве источника информации могут использоваться РОН, константа или значение из памяти;

П – приемник информации. В качества приемника может выступать только РОН.

Возможные варианты команды IMUL в двухоперандном виде:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| П |  | П |  | И |
| 16-ти разрядный РОН | = | 16-ти разрядный РОН | \* | 16-ти разрядный РОН или ЯП |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН | \* | 32-ух разрядный РОН или ЯП |
| 16-ти разрядный РОН | = | 16-ти разрядный РОН | \* | 8-ми разрядная константа1 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН | \* | 8-ми разрядная константа2 |
| 16-ти разрядный РОН | = | 16-ти разрядный РОН | \* | 16-ми разрядная константа |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН | \* | 32-ми разрядная константа |

1 - 8-ми разрядная константа, перед умножение знакорасширяется до 16-ти разрядов.

2 - 8-ми разрядная константа, перед умножение знакорасширяется до 32-ух разрядов.

1. ий формат команды (трехоперандный): IMUL П,И1,И2

где И1 – первый источник информации. В качестве И1 могут использоваться РОН или значение из памяти;

И2 – второй источник информации. В качестве И2 может использоваться только константа;

П – приемник информации. В качества приемника может выступать только РОН.

Возможные варианты команды IMUL в трехоперандном виде:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| П |  | И1 |  | И2 |
| 16-ти разрядный РОН | = | 16-ти разрядный РОН или ЯП | \* | 8-ми разрядная константа1 |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН или ЯП | \* | 8-ми разрядная константа2 |
| 16-ти разрядный РОН | = | 16-ти разрядный РОН или ЯП | \* | 16-ми разрядная константа |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН или ЯП | \* | 16-ми разрядная константа3 |
| 32-ух разрядный РОН | = | 32-ух разрядный РОН или ЯП | \* | 32-ми разрядная константа |

1 - 8-ми разрядная константа, перед умножение знакорасширяется до 16-ти разрядов. 2 - 8-ми разрядная константа, перед умножение знакорасширяется до 32-ух разрядов. 3 - 16-ти разрядная константа, перед умножение знакорасширяется до 32-ух разрядов.

Когда используется однооперандная форма команды IMUL, то результат умножения доступен, даже если установлен флаг переполнения, так как РОН результата в два раза больше по размеру, чем множимое и множитель. Этого достаточно, чтобы сохранить любой возможный результат.

Для форм с двумя и тремя операндами возможна потеря старших значащих битов результата. Если это происходит, то флаги CF и OF устанавливаются.

**Команда DIV, IDIV**

Формат команды: DIV И

IDIV И

где И – источник информации. В качестве источника информации могут использоваться РОН или ячейки памяти.

Команда DIV выполняет операцию целочисленного беззнакового деления. Команда IDIV выполняет операцию целочисленного деления с учетом знака. При выполнении и команды DIV, и команды IDIV используются неявные операнды:

* если И 8-ми разрядный, то AX делится на И. Целочисленное частное сохраняется в AL, целочисленный остаток в AH;
* если И 16-ти разрядный, то DX:AX делится на И. Целочисленное частное сохраняется в AX, целочисленный остаток в DX;
* если И 32-ух разрядный, EDX:EAX делится на И. Целочисленное частное сохраняется в EAX, целочисленный остаток в EDX.

Значение флагов OF,SF,ZF,AF,PF,CF неопределенно, т.е. стоит считать их значения после выполнение команды случайным.

Если результирующее частное слишком велико, чтобы поместиться по назначению, или если делитель равен нулю, то генерируется исключительная ситуация, которая отслеживается на уровне операционной системы, однако при этом программа может продолжить свое выполнение.

Следующие четыре команды очень похожи на команду MOVSX. Однако, команда MOVSX является универсальной и может использовать любые РОН и ячейки памяти в качестве своих операндов. Рассматриваемые же здесь команды не имеют входных операндов, операнды используются неявно. Это дает возможность экономить память программ и, более того, эти команда используют те же самые РОН, что и команды умножения и деления, а, значит, использование их в связке повышает эффективность программного кода.

**Команда CBW**

Формат команды: CBW

Команда CBW знаково расширяет байт, находящийся в регистре AL, до слова и сохраняет результат в регистре AX.

**Команда CWD**

Формат команды: CWD

Команда CWD знаково расширяет слово, находящиеся в регистре AX, до двойного слова и сохраняет результат в паре регистров DX:AX.

**Команда CDQ**

Формат команды: CDQ

Команда CDQ знаково расширяет двойное слово, находящееся в EAX, в учетверенное слово, и результат помещает в пару регистров EDX:EAX.

**Команда CWDE**

Формат команды: CWDE

Команда CWDE знаково расширяет слово, находящееся в регистре AX, в двойное слово, и результат помещает в регистр EAX.

**Примерные контрольные вопросы и задачи**

1. Чему будут равны регистры CH, CL, CX, BL, BH после выполнения следующего фрагмента программы:

MOV CL,0 MOV BL,0

MOV BL,$32 MOV CL,24 ADD BL,CL

1. Чему будут равны регистры AH, AL, DX, AL после выполнения следующего фрагмента программы:

MOV EAX,0 MOV EDX,0 MOV AX,$32 MOV DX,2 SUB AX,DX

1. Чему будут равны регистры EAX, ECX после выполнения следующего фрагмента программы:

MOV EAX,0 MOV EDX,0 MOV AX,$3D MOV CL,2 MUL CL

1. Чему будут равны регистры EAX, ECX, DX, AX после выполнения следующего фрагмента программы:

MOV EAX,0 MOV EDX,0 MOV AX,200 MOV CL,2 DIV CL

1. Чему будут равны регистры EAX, ECX, DX, AX после выполнения следующего фрагмента программы:

MOV EAX,0 MOV EDX,0 MOV AX,$200 MOV CX,$2 DIV CX

1. Чему будет равен регистры EAX после выполнения следующего фрагмента программы:

MOV AX,243 CWDE

# Лабораторная работа №3. Команды передачи управления

ними.

**Цель работы**

Изучение команд передачи управления и получение навыка работы с

**Постановка задачи**

* 1. С помощью процедуры *Random* сформировать случайное целое число в диапазоне от 0 до 65535;
  2. Четное число (четное с точки зрения математики) записать в ЯП (ячейку памяти), начиная с нулевого элемента массива *data*. Нечетное число (нечетное с точки зрения математики) записать в ЯП, начиная с 128 элемента массива *data*. Все числа, которые больше или равны 50000 записать в ЯП, начиная с 256 элемента массива *data*. Все числа, которые меньше 10000 записать в ЯП, начиная с 384 элемента массива *data*;
  3. Повторять п.1-п.2, пока число распределенных данных по любому из признаков не превысит 127.

**Содержание отчета**

Отчет по лабораторной работе должен содержать номер, название, цель лабораторной работы, задание на лабораторную работу, текст программы на языке ассемблера, написанного от руки при подготовке, распечатки формы в активном состоянии с результатами выполнения лабораторной работы в виде дампа памяти и фрагмента программы, содержащей ассемблерную вставку, а также вывод по работе.

Пояснения к выполнению лабораторной работы:

1. В самом начале программы выполнить процедуру *Randomize*;
2. Ассемблерную вставку (если это необходимо) разбить на две части, до применения процедуры *Random* и после ее применения;
3. Перед применением процедуры *Random* сохранить необходимые РОН в стеке, а после ее применения восстановить их;
4. Использование процедуры *Random* выглядит следующим образом:

*A := Random(Range*);

где *А* - переменная типа *Word*.

При этом переменная *А* станет равной: *0 ≤ А < Range*.

**Краткие теоретические сведения**

К командам передачи управления относятся:

JMP (Jump) – Прыгнуть, перескочить, скачок;

Jcc (Jump Condition Code) – Прыгнуть по коду условия. Данная

расшифровка аббревиатуры команды является традиционной и официальной.

Однако для понимания сути команды можно привести другую расшифровку: Jump if the Condition is true or Continue if it is false – прыгнуть, если условие истинно или продолжить, если оно ложно;

LOOP (Loop) – Цикл;

CALL (Call) – Вызов;

RET (Return) – Возврат;

CMP (Compare) – Сравнить;

CMPXCHG (Compare and eXchange) – Сравнить и обменять;

CMPXCHG8B (Compare and eXchange 8 Bytes) – Сравнить и обменять 8 байтов;

**Команда JMP**

Формат команды: JMP Операнд

В качестве операнда может выступать непосредственный адрес (число) или РОН, или ячейка памяти, содержащие адрес кода программы в памяти, с которого должно продолжиться выполнение программы.

Программа, которую выполняет микропроцессор, храниться в оперативной памяти. После компиляции, все команды превращаются в шестнадцатеричные коды, которые последовательно хранятся в памяти. Каждому коду программы соответствует адрес в памяти, так как память микропроцессорных систем можно представить в виде одномерного массива, где каждому элементу массива присвоен номер (адрес). Это прекрасно видно на рисунке 6 в первой его области: первый столбец – адреса ячеек памяти, второй столбец (через пробел от первого) – шестнадцатеричные коды программы, третий столбец – соответствующая кодам из второго столбца мнемоника команд. Команда JMP указывает микропроцессору, начиная с какого адреса следует дальше выполнять команды.

Команда JMP может осуществлять четыре различных типа перехода:

* переход типа *short* (короткий переход) – если адрес перехода находиться в пределах -128…+127 байт от команды, следующей за командой JMP;
* переход типа *near* (ближний переход) – если адрес перехода находиться в том же сегменте памяти, что и команда JMP;
* переход типа *far* (дальний переход) – если адрес перехода находиться в другом сегменте. Дальний переход может выполняться и в тот же самый сегмент при условии, что в сегментной части операнда указано число, совпадающее с текущим значением *CS*;
* переход с переключением задачи – передача управления другой задаче в многозадачной среде.

При выполнении перехода типа *short* и *near* команда JMP фактически преобразовывает значение регистра EIP, изменяя тем самым смещение

следующей исполняемой команды относительно начала сегмента кода. Если операнд – регистр или ячейка памяти, то его значение просто копируется в EIP, как если бы это была команда MOV. Если операнд команды JMP – непосредственно указанное число, то это число воспринимается как число со знаком и его значение суммируется с содержимым EIP, приводя к относительному переходу. В ассемблерных программах в качестве операнда обычно указывают имена меток. Программисту сложно каждый раз вычислять конкретное значение адреса перехода и указывать его в команде. Абсолютно логично возложить это на компилятор. На этапе компиляции вычисляются адреса всех меток и полученные значения записываются во все команды, использующие эти метки, в виде относительного смещения.

**Команда Jсс**

Формат команды: Jсс метка

В качестве метки может выступать непосредственный адрес для перехода, а так же РОН или ячейка памяти, содержащие адрес для перехода.

Под аббревиатурой Jсс понимается набор команд (см. таблицу 5), осуществляющий условный переход типа short или near. Команды с мнемоникой Jcc НЕТ. Вместо "сс" после J следуют буквы, определяющие условие перехода. Условиями перехода являются состояния флагов, определенных в команде. Если условие выполняется, то управление передается по адресу, указанному в команде. Если условие не выполняется, то микропроцессор продолжает выполнение программы с команды, следующей за командой условного перехода.

Чаще всего, команда из набора Jcc используется сразу после команды CMP. В этом случае условие приобретает формулировку, соответствующую отношению между операндами в команде CMP. В таблице 5 представлены варианты команды Jcc. Например, если операнды CMP были равны, то команда JE, выполненная сразу после CMP, осуществит переход, так как при вычитании двух равных операндов получится нуль, значит, установится флаг нуля ZF, что является условием перехода в команде JE.

Слова «выше» и «ниже» в таблице 5 относятся к сравнению беззнаковых чисел. Слова «больше» и «меньше» относятся к сравнению чисел со знаком.

Таблица 5

|  |  |  |  |
| --- | --- | --- | --- |
| Мнемоника  команды | Описание | Перевод описания | Условие перехода |
| JA | Jump if above | Переход, если выше | CF=0 и ZF=0 |
| JAE | Jump if above or equal | Переход, если выше или равно | СF=0 |
| JB | Jump if below | Переход, если ниже | СF=1 |
| JBE | Jump if below or equal | Переход, если ниже или равно | СF=1 или ZF=1 |
| JC | Jump if carry | Переход, если перенос | СF=1 |
| JCXZ | Jump if CX is zero | Переход, если регистр CX=0 | CX=0 |
| JECXZ | Jump if ECX is zero | Переход, если ECX=0 | ECX=0 |
| JE | Jump if equal | Переход, если равно | ZF=1 |
| JG | Jump if greater | Переход, если больше | ZF=0 и SF=OF |
| JGE | Jump if greater or equal | Переход, если больше или равно | SF=OF |
| JL | Jump if less | Переход, если меньше | SF≠OF |

|  |  |  |  |
| --- | --- | --- | --- |
| JLE | Jump if less or equal | Переход, если меньше или равно | ZF=1 или SF≠OF |
| JNA | Jump if not above | Переход, если не выше | CF=1 или ZF=1 |
| JNAE | Jump if not above or equal | Переход, если не выше или равно | СF=1 |
| JNB | Jump if not below | Переход, если не ниже | СF=0 |
| JNBE | Jump if not below or equal | Переход, если не ниже или равно | CF=0 и ZF=0 |
| JNC | Jump if not carry | Переход, если нет переноса | CF=0 |
| JNE | Jump if not equal | Переход, если не равно | ZF=0 |
| JNG | Jump if not greater | Переход, если не больше | ZF=1 или S≠OF |
| JNGE | Jump if not greater or equal | Переход, если не больше или равно | SF≠OF |
| JNL | Jump if not less | Переход, если не меньше | SF=OF |
| JNLE | Jump if not less or equal | Переход, если не меньше или равно | ZF=0 и SF=OF |
| JNO | Jump if not overflow | Переход, если нет переполнения | ОF=0 |
| JNP | Jump if not parity | Переход, если нечетно (не математически) | PF=0 |
| JNS | Jump if not sign | Переход, если результат положителен | SF=0 |
| JNZ | Jump if not zero | Переход, если не нуль | ZF=0 |
| JO | Jump if overflow | Переход, если переполнение | ОF=1 |
| JP | Jump if parity | Переход, если четно (не математически) | PF=1 |
| JS | Jump if sign | Переход, если результат отрицательный | SF=1 |
| JZ | Jump if zero | Переход, если ноль | ZF=1 |

**Группа команд LOOP**

Команды этой группы организуют цикл, используя в качестве счетчика числа итераций регистр **ECX** (другое название регистра ECX: *counter* – счетчик). В регистр **ECX** должно быть предварительно занесено необходимое число итераций. Команды этой группы реализуют классический цикл с уменьшающимся на 1 на каждой итерации счетчиком и постусловием проверки счетчика на нулевое значение. При уменьшении счетчика флаги процессора не меняются!

**Команда LOOP**

Формат команды: **LOOP метка\_перехода**

Команда LOOP уменьшает значение регистра ECX на 1 и выполняет переход типа *short* на метку (переход не может быть дальше расстояния в

-128…+127 байт от команды LOOP), если ECX не равен нулю.

**Команды LOOPE и LOOPZ**

Формат команды: **LOOPE метка\_перехода**

Формат команды: **LOOPZ метка\_перехода**

Команда LOOPE (LOOPZ) аналогична команде LOOP, только добавляется проверка флага ZF на равенство единице (ZF=1) по логической связке И с условием неравенства ECX нулю. Применяется данная команда в случае, когда нужно досрочно выйти из цикла.

**Команды LOOPNE и LOOPNZ**

Формат команды: **LOOPNE метка\_перехода**

Формат команды: **LOOPNZ метка\_перехода**

Команда LOOPNE (LOOPNZ) аналогична команде LOOP, только добавляется проверка флага ZF на равенство нулю (ZF=0) по логической связке И с условием неравенства ECX нулю.

Выполнение этих команд не влияет на флаги! Так как команды группы LOOP не влияют на флаги, их удобно использовать вместе с командами, которые в результате своей работы меняют значение флага ZF, например, с командой сравнения CMP или с командой вычитания SUB.

Недостаток всех команд цикла в том, что они реализуют только короткие переходы. Для работы с длинными циклами используются команды условного и безусловного перехода, описанные выше.

**Команда CALL**

Формат команды: **CALL** *операнд*

Сохраняет текущий адрес в стеке и передает управление по адресу, указанному в операнде.

Операндом может быть непосредственное значение адреса (или метка), регистр или ячейка памяти, содержащие адрес перехода.

Часто возникает необходимость выполнения специфической операции в разных местах программы. В этом случае в каждом месте выполнения специфической операции надо продублировать последовательность команд, исполняющих эту операцию. Конечно, такое дублирование раздражает и отнимает много времени. Кроме того, оно значительно удлиняет программу.

На самом деле можно избежать дублирования, если определите повторяющуюся последовательность команд как процедуру. Процедура (или, как часто говорят, подпрограмма) представляет собой совокупность команд, которая написана один раз, но может быть исполнена по мере необходимости в любом месте программы.

Процесс передачи управления из основной части программы в процедуру называется вызовом, т.е. процедура вызывается. При вызове процедуры микропроцессор исполняет ее команды, а затем возвращается к тому месту, откуда был сделан вызов.

Возникают два вопроса: как вызвать процедуру и как микропроцессор возвращает управление в нужное место программы? Таким образом, команды, обеспечивающие исполнение процедур, должны выполнять три функции:

* 1. Обеспечить сохранение содержимого указателя команд EIP. Когда процедура исполнена, находившийся в этом указателе адрес используется микропроцессором для возврата к месту вызова. Будем называть его **адресом возврата**;
  2. Заставить микропроцессор начать исполнение процедуры;
  3. Использовать сохраненное содержимое указателя команд EIP для возврата в программу и обеспечить продолжение ее исполнения с этого места.

Все эти функции выполняются двумя командами: CALL (call a procedure - вызвать процедуру) и RET (return from procedure - возвратиться из процедуры).

Команда CALL осуществляет функции запоминания адреса возврата и передачи управления процедуре. Она помещает в стек адрес возврата. Объем данных запоминаемого адреса зависит от типа перехода (см. команду JMP).

Если в качестве адреса перехода указано только смещение, считается, что адрес расположен в том же сегменте, что и команда CALL. При этом выполняется ближний вызов процедуры, т.е. в том же сегменте кода. Процессор помещает значение регистра EIP, соответствующее команде следующей за командой CALL, в стек и загружает в EIP значение операнда, осуществляя тем самым передачу управления:

PUSH EIP

EIP=EIP+*смещение\_к\_нужной\_команде*

Если операнд CALL – регистр или переменная, то его значение рассматривается как абсолютное смещение, если операнд – ближняя метка в программе, то ассемблер указывает ее относительное смещение. Чтобы выполнить дальний CALL в реальном режиме процессор помещает в стек значения регистров CS и EIP и осуществляет дальний переход аналогично команде JMP.

**Команда RET**

Формат команды: **RET[** *Im16***]**

Команда RET заставляет микропроцессор возвратиться из процедуры в программу, вызвавшую эту процедуру делая это "откатом" всего, что сделала команда CALL. Команда RET обязательно должна быть последней командой процедуры, исполняемой микропроцессором. (Это не значит, что команда RET должна стоять в конце процедуры - она лишь исполняется последней.)

Команда RET извлекает из стека адрес возврата, записанный туда командой CALL, и загружает его в счетчик команд EIP.

Операнд для RET не обязателен, но, если он присутствует, то после считывания адреса возврата из стека будет удалено указанное количество байтов – это нужно, если при вызове процедуры ей через стек передавались параметры.

**Команда CMP**

Формат команды: CMP О1,О2

где О1 – первый операнд. В качестве первого операнда могут использоваться РОН или значение из памяти.

О2 – второй операнд. В качестве второго операнда могут использоваться РОН, число или значение из памяти.

*П и И не могут быть одновременно значениями из памяти!!!*

Команда CMP сравнивает содержимое первого и второго операндов путем вычитания второго операнда из первого, результат никуда не сохраняется, а устанавливаются лишь флаги. Единственным результатом этой

команды является изменение флагов CF,OF,SF,ZF,AF,PF в соответствии с результатом вычитания. В основном эта команда используется вместе с командами условного перехода.

**Команда CMPXCHG**

Формат команды: CMPXCHG О1,О2

Если О1 8-ми разрядный РОН или ячейка памяти, то О2 должен быть 8- ми разрядным РОН. При этом, сравниваются значения регистра AL и О1 путем вычитания из AL значения О1 и, если значения равны, то ZF=1 и О1=О2, иначе ZF=0 и AL=О1.

Если О1 16-ти разрядный РОН или ячейка памяти, то О2 должен быть 16- ти разрядным РОН. При этом, сравниваются значения регистра AX и О1 путем вычитания из AX значения О1 и, если значения равны, то ZF=1 и О1=О2, иначе ZF=0 и AX=О1.

Если О1 32-ух разрядный РОН или ячейка памяти, то О2 должен быть 32- ух разрядным РОН. При этом, сравниваются значения регистра EAX и О1 путем вычитания из EAX значения О1 и, если значения равны, то ZF=1 и О1=О2, иначе ZF=0 и EAX=О1.

Как и в команде CMP, результаты вычитания никуда не сохраняется, а устанавливаются лишь флаги СF,OF,SF,ZF,AF,PF в соответствии с результатом вычитания.

**Команда CMPXCHG8B**

Формат команды: CMPXCHG8B О1

где О1 – первый операнд. В качестве первого операнда может использоваться только 64-х разрядное значение из памяти.

Команда CMPXCHG8B сравнивает содержимое пары регистров EDX:EAX и первого операнда путем вычитания первого операнда из пары регистров EDX:EAX. Результат вычитания никуда не сохраняется. Если значения равны, то ZF=1 и О1=ECX:EBX, иначе ZF=0 и EDX:EAX=О1. Флаги СF,OF,SF,AF,PF устанавливаются в соответствии с результатом вычитания.

**Примерные контрольные вопросы и задачи**

1. Влияет ли результат команды LOOPE на флаги?
2. Какой регистр выступает в роли счетчика при организации циклов с помощью команды LOOP?
3. Какой флаг учитывает команда LOOPNZ при переходе к началу цикла? 4.Какой флаг учитывает команда LOOPE при выходе из цикла?
4. Чем отличается условный переход от безусловного? Какие команды осуществляют условный переход?
5. Какой тип переходов (ближний или дальний) реализуются командами LOOP,LOOPE,LOOPNE?
6. Сохраняет ли команда безусловного перехода JMP информацию о точке возврата?
7. Какие значения примут регистры EAX, ECX после выполнения следующего фрагмента программы:

MOV EAX,$2 MOV ECX, 10

@LOOP\_START: ADD EAX, ECX

LOOP @LOOP\_START

# Лабораторная работа №4. Логические команды и команды манипулирования битами

ними.

**Цель работы**

Изучение команд передачи управления и получение навыка работы с

**Постановка задачи**

1. Занести в память 32 разрядное шестнадцатеричное число в соответствии с вариантом из таблицы 6;
2. Подсчитать количество нулей и единиц в данном числе двумя разными способами;
3. Подсчитать количество парных нулей и парных единиц в данном числе;
4. В младшем байте числа обменять между собой биты 0-7, 1-6, 2-5, 3-4.

Результаты каждого пункта сохранить в памяти и распечатать.

**Содержание отчета**

Отчет по лабораторной работе должен содержать номер, название, цель лабораторной работы, задание на лабораторную работу, текст программы на языке ассемблера, написанного от руки при подготовке, распечатки формы в активном состоянии с результатами выполнения лабораторной работы в виде дампа памяти и фрагмента программы, содержащей ассемблерную вставку, а также контрольный пример и выводы по работе.

**Варианты заданий**

Таблица 6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер  варианта | Число (HEX) | Номер  варианта | Число(HEX) | Номер  варианта | Число(HEX) |
| 1 | **DEFBCDEF** | 21 | **23456ABC** | 41 | **4467ADEC** |
| 2 | **264535АВ** | 22 | **34578ABC** | 42 | **BCDE5587** |
| 3 | **DFAC3421** | 23 | **DABC47AE** | 43 | **75421DDE** |
| 4 | **1254FEDB** | 24 | **BDCE4578** | 44 | **23794685** |
| 5 | **12546FD1** | 25 | **58697EFF** | 45 | **CEF59764** |
| 6 | **FDCB5478** | 26 | **22546781** | 46 | **162894BB** |
| 7 | **ABCDEF45** | 27 | **FADCDAD5** | 47 | **FEADCBDE** |
| 8 | **AAABCD45** | 28 | **FEAD45AD** | 48 | **CCBDEA43** |
| 9 | **BC41424D** | 29 | **12451346** | 49 | **1457BCEC** |
| 10 | **1245BC41** | 30 | **ADAECB44** | 50 | **CBDEA785** |
| 11 | **DA2AD355** | 31 | **578FAEDB** | 51 | **CBDAE145** |
| 12 | **47AC2EFA** | 32 | **ABCD1542** | 52 | **CBAFE432** |
| 13 | **BCD45AE1** | 33 | **1679EADB** | 53 | **CBFAE342** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14 | **14523678** | 34 | **1346FAED** | 54 | **1678ADEC** |
| 15 | **ABCDEF11** | 35 | **ABABDFAB** | 55 | **47215F57** |
| 16 | **BCDEF44A** | 36 | **4678FEAB** | 56 | **ADCBE875** |
| 17 | **1245DBCC** | 37 | **CB45AE2A** | 57 | **BCDE22EA** |
| 18 | **ABE4578A** | 38 | **25467BCA** | 58 | **4875ADCB** |
| 19 | **8754AD4F** | 39 | **12478BCE** | 59 | **BCDEFA2B** |
| 20 | **12345678** | 40 | **BCDAE14A** | 60 | **17563BCB** |

**Краткие теоретические сведения**

Различают команды, выполняющие логические операции и команды манипулирования битами. Команды выполняют операции с битами байта, слова или двойного слова, поэтому еще называются побитовыми (поразрядными) командами.

Теоретической базой для логических операций является булева алгебра, которая впервые была исследована Дж. Булем (1815-1864). Она базируется на высказываниях. Высказывание – это законченное предложение, о котором можно определенно сказать, что его содержание истинно или ложно. Операции булевой алгебры определены для таких высказываний. Всевозможные наборы входных значений и значений булевой функции на этих наборах входных значений составляют таблицу истинности данной булевой функции.

К командам, выполняющим логические операции, и командам манипулирования битами относятся:

AND – Операция логического умножения «И»;

OR – Операция логического сложения «ИЛИ»; XOR (eXclusive OR) – Операция «исключающее ИЛИ»;

NOT – Операция двоичной инверсии; SAL (Shift Arithmetic Left) – Арифметический сдвиг влево; SAR (Shift Arithmetic Right) – Арифметический сдвиг вправо; SHL (SHift Left) – Сдвиг влево;

SHR (SHift Right) – Сдвиг вправо;

SHLD (SHift Left Double) – Сдвиг влево двойной точности; SHRD (SHift Right Double) – Сдвиг вправо двойной точности; ROL (ROtate Left) – Циклический сдвиг влево;

ROR (ROtate Right) – Циклический сдвиг вправо;

RLC (Rotate through Carry Left) – Циклический сдвиг влево через флаг

CF;

RRC (Rotate through Carry Right) – Циклический сдвиг вправо через

флаг CF;

BT (Bit Test) – Переместить бит для тестирования; BTS (Bit Test and Set) – Протестировать бит и установить его;

BTC (Bit Test and Complement) – Протестировать бит и инвертировать

его;

BTR (Bit Transfer and Reset) – Протестировать бит и сбросить его;

BSF (Bit Scan Forward) – Сканировать биты в прямом

направлении;

BSR (Bit Scan Reverse) – Сканировать биты в обратном

направлении;

|  |  |  |
| --- | --- | --- |
| TEST |  | – Протестировать операнд; |
| CLC | (Clear CF) | – Очистить флаг CF; |
| CLD | (Clear DF) | – Очистить флаг DF; |
| CMC | (Complement CF) | – Инвертировать флаг CF; |
| STC | (Set CF) | – Установить флаг CF; |
| STD | (Set DF) | – Установить флаг DF; |

**Команды AND,OR,XOR** Формат команды: AND П,И Формат команды: OR П,И

Формат команды: XOR П,И

Команды выполняют побитовые логические операции над приемником П, в качестве которого могут выступать регистр или ячейка памяти, и источником И, в качестве которого могут выступать регистр, ячейка памяти или константа. Источник и приемник не могут быть одновременно ячейками памяти. Результат операции помещается в приемник П. Таблицы истинности для каждой функции представлены в таблице 7.

таблица 7.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **AND** |  |  |  | **OR** |  |  |  | **XOR** |  |  | **NOT** | |
| Логическое умножение | | | Логическое сложение | | | Логическое  исключающее ИЛИ | | | Логическое отрицание | |
| X | Y | Вых. | X | Y | Вых. | X | Y | Вых. | Х | Вых. |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |  | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Один из распространенных вариантов применения команды AND для выполнения операции маскирования или наложения маски. Как и в случае с обычной, театральной, маской, сквозь нее видно только там, где есть прорези. Если нам необходимо, чтобы в каком-либо операнде одни значения битов под конкретными номерами сохранились, а другие нет (обнулились), то мы выполним команду AND, в которой в качестве второго операнда будет такое значение, в котором на месте битов, значения которых надо сохранить, будут стоять «1», а во всех остальных нули.

Команду OR чаще всего используют для выборочной установки отдельных битов без изменения значений остальных битов. Это достигается за счет того, что команда OR выполняет логическое сложение, а значит, сложение

с нулем не меняет значение бита, а сложение с «1» установит бит, вне зависимости от его предыдущего значения.

Команда XOR используется в разных логических операциях. Иногда, чаще используют две ее особенности. Если выполнить команду XOR, в которой приемник и источник равны, то результатом команды будет нуль. Вторая особенность, если дважды выполнить команду XOR с одним и тем же источником, то приемник станет равным своему первоначальному значению.

Флаги, которые устанавливаются в результате этой команды, приведены в таблице 8.

Таблица 8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Синтаксис** | **Биты регистра флагов** | | | | |
| **OF** | **SF** | **ZF** | **PF** | **CF** |
| **AND** П, И | 0 | + | + | + | 0 |
| **TEST** П,И | 0 | + | + | + | 0 |
| **OR** П,И | 0 | + | + | + | 0 |
| **XOR** П,И | 0 | + | + | + | 0 |
| **NOT** П | - | - | - | - | - |

**Команда NOT**

Формат команды: NOT П

Команда NOT выполняет операцию побитовой инверсии приемника П, в качестве которого могут выступать РОН или ячейка памяти.

**Команда TEST**

Формат команды: TEST П,И

Команда TEST эквивалентна команде AND, т.е. побитово осуществляет операцию логического умножения над приемником и источником и устанавливает флаги (см.таблицу 8), но результат операции никуда не записывается. Команда TEST, так же как и CMP, используется в основном в сочетании с командами условного перехода.

**Команды SAL, SAR, SHL, SHR, ROL, ROR, RLC, RRC**

Формат команды: SAL П,И Формат команды: SAR П,И Формат команды: SHL П,И Формат команды: SHR П,И Формат команды: ROL П,И Формат команды: ROR П,И Формат команды: RLC П,И Формат команды: RRC П,И

Мнемоника Графическое изображение выполняемых действий

Таблица 9.

Влияние на флаги

OF SF ZF PF CF

SAL



CF

|  |  |  |
| --- | --- | --- |
| бит n | бит  n-1 | бит  n-2 |

|  |  |  |
| --- | --- | --- |
| бит 2 | бит 1 | бит 0 |

SAR



бит

n-2

бит

n-1

бит n



|  |  |  |
| --- | --- | --- |
| бит 2 | бит 1 | бит 0 |

SHL



CF

|  |  |  |
| --- | --- | --- |
| бит n | бит  n-1 | бит  n-2 |

|  |  |  |
| --- | --- | --- |
| бит 2 | бит 1 | бит 0 |

SHR 0



|  |  |  |
| --- | --- | --- |
| бит n | бит  n-1 | бит  n-2 |

|  |  |  |
| --- | --- | --- |
| бит 2 | бит 1 | бит 0 |

ROL



бит 0

бит 1

бит 2

бит

n-2

бит

n-1

бит n

CF

0 ? + + + +

? + + + +

CF

0 ? + + + +

? + + + +

CF

+ - - - +



бит 0

бит 1

бит 2

бит

n-2

бит

n-1

бит n

CF

ROR + - - - +

RLC 



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | CF |  | бит | бит |  |  |  |  |
|  | бит n | n-1 | n-2 | бит 2 | бит 1 | бит 0 |  |
|  | | | | | | | |

RRC



CF

бит 0

бит 1

бит 2

бит

n-2

бит

n-1

бит n

+ - - - +

+ - - - +

В данных командах происходит сдвиг битов, в соответствие с таблицей 9, приемника П, в качестве которого может выступать РОН или ячейка памяти, на число двоичных разрядов, указанных в источнике И, в качестве которого может выступать число 1, байтовая константа или регистр CL.

Одно из применений этих команд состоит в том, что они позволяют умножать (при сдвиге влево) или делить (при сдвиге вправо) значение приемника на 2**n**, где n – количество сдвигов. При этом скорость выполнения команд сдвига намного больше, чем скорость выполнения команд умножения или деления. Например, следующий фрагмент программы:

MOV BX,AX SHL AX,2 ADD AX,BX SHL AX,1

выполнит умножение содержимого регистра AX на 10 в ОДИННАДЦАТЬ раз быстрее, чем команда MUL.

**Команды SHLD, SHRD**

Формат команды: SHLD П,И1,И2 Формат команды: SHRD П,И1,И2

В данных командах происходит сдвиг битов, в соответствие с таблицей 10, приемника П, в качестве которого могут выступать только 16-ти и 32-ух разрядные РОН или ячейки памяти на число двоичных разрядов, указанных во втором источнике И2, в качестве которого может выступать байтовая константа или регистр CL. Источник И1 содержит биты, которые будут заноситься в приемник по мере смещения, т.е. будет служить источником битов, при этом, в приемник П занесется столько битов из И1, сколько будет указано в И2. В самом И1 биты не сдвигаются и к концу операции в И1 будет храниться первоначальное значение. В качестве И1 может использоваться только РОН, причем такой же размерности, что и П.

Таблица 10.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Мнемоника | Графическое изображение выполняемых действий | | | | | | | | | | | | | | | | | Влияние на флаги | | | | |
| OF | SF | ZF | PF | CF |
| SHLD | CF |  | бит n | бит  n-1 | бит  n-2 | П | бит 2 | бит 1 | бит 0 |  | бит n | бит  n-1 | бит  n-2 | И1 | бит 2 | бит 1 | бит 0 | ? | + | + | + | + |
| SHRD | бит n | бит  n-1 | бит  n-2 | И1 | бит 2 | бит 1 | бит 0 |  | бит n | бит  n-1 | бит  n-2 |  | бит 2 | бит 1  П | бит 0 |  | CF | ? | + | + | + | + |

**Команда BT**

Формат команды: BT И1,И2

Команда BT считывает во флаг CF значение бита из первого источника И1, в качестве которого может выступать только 16-ти и 32-ух разрядный РОН или ячейка памяти, с номером, указанным во втором источнике И2, в качестве которого может выступать только 16-ти и 32-ух разрядный РОН или байтовая константа. Значение второго источника берется по модулю 32.

После выполнения команды BT флаг CF равен значению считанного бита, а флаги OF, SF, ZF, AF и PF не определены.

**Команды BTS, BTR, BTC** Формат команды: BTS И1,И2 Формат команды: BTR И1,И2

Формат команды: BTC И1,И2

Эти команды повторяют действие команды BT, т.е. считывают во флаг CF значение бита из первого источника И1 с номером, указанным во втором источнике И2, но после этого, соответственно, считанный бит в приемнике П устанавливается в 1 (BTS), сбрасывается в 0 (BTR) и инвертируется (BTC).

После выполнения команд BTS, BTR и BTC флаг CF равен значению считанного бита до его изменения в результате действия команды, а флаги OF, SF, ZF, AF и PF не определены.

**Команды BSF, BSR** Формат команды: BSF П,И Формат команды: BSR П,И

BSF сканирует источник И, в качестве которого может выступать только 16-ти и 32-ух разрядный РОН или ячейка памяти, начиная с самого младшего бита (т.е. слева направо), и записывает в приемник П, в качестве которого может выступать только РОН такой же разрядности, что и источник И, номер первого встретившегося бита, равного 1. Команда BSR эквивалентна команде BSF, но сканирует источник И, начиная с самого старшего бита. Например, если источник И равен 0000'0000'0000'01002, то и команда BSF, и команда BSR возвратит 2.

Если весь источник равен нулю, значение приемника не определено и флаг ZF устанавливается в 1, иначе ZF сбрасывается. Флаги OF, SF, CF, AF и PF не определены.

**Команды CLC, CLD, CMC, STC, STD**

Формат команды: CLC Формат команды: CLD Формат команды: CMC Формат команды: STC Формат команды: STD

Команда CLC сбрасывает флаг CF (CF=0). Команда CLD сбрасывает флаг DF (DF=0). Команда CMC инвертирует флаг CF (CF= CF ).

Команда STC устанавливает флаг CF (CF=1). Команда STD устанавливает флаг DF (DF=1).

**Примерные контрольные вопросы и задачи**

1. Приведите фрагменты программ на языке ассемблера, выполняющие следующие действия: сдвиг СХ на 2 бита вправо; сдвиг АХ на 2 бита влево; приравнивание значения 426 регистру СХ; сравнение значения переменной BYTE1 типа Byte и числа 2516.
2. Полагая, что DL содержит 1111'00012, а переменная BOOL типа Byte содержит 1110'00112, определите значение регистра DL после выполнения каждой отдельно взятой команды:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| а) | AND | DL,BOOL | b) | OR DL,BOOL |
| c) | XOR | DL,BOOL | d) | AND DL,0 |
| e) | XOR | DL,$FF | f) | NOT DL |

После выполнения каких команд будет установлен флаг ZF?

1. Полагая, что DX содержит значение 1011'1001'1011'10012, определите, какое значение будет содержать DX после выполнения следующих отдельно взятых инструкций:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a) | SHL | DL,1 | b) | SHL | DX,2 | c) | SHR DX,1 |
| d) | SAR | DX,2 | e) | SAL | DH,3 | f) | ROR DX,3 |
| g) | ROR | DL,3 | h) | RCL | DX,17 |  |  |

1. Чему будет равен регистр АX в результате следующего фрагмента программы:

MOV CX,$0C BSF AX,CX

1. Чему будет равен регистр СX в результате следующего фрагмента программы (М – переменная типа Word):

MOV CX,$2C MOV M,CX BSF CX,M

1. Как приравнять одной командой регистр ECX к нулю десятью разными способами (можно использовать команды из всех лабораторных работ)? Начальное значение регистра ECX и флагов случайное.

# Список использованной и рекомендуемой литературы

* 1. И.И.Шагурин, Е.М.Бердышев Процессоры семейства INTEL P6, Pentium II, Pentium III, Celeron и др. Архитектура, программирование, интерфейс. Справочник «Горячия линия – Телеком», 2000;
  2. А.А.Ровдо Микропроцессоры от 8086 до Pentium III Xeon и AMD- K6-3 «ДМК», 2000;
  3. Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения – 2-е изд., испр. и доп. – СПб.: ООО «ДиаСофтЮП», 2002. – 656с.
  4. Intel® 64 and IA-32 Architectures Software Developer’s Manual. Volume 2A: Instruction Set Reference, A-M [www.intel.com](http://www.intel.com/) Order Number: 253666-028US
  5. Intel® 64 and IA-32 Architectures Software Developer’s Manual. Volume 2B: Instruction Set Reference, N-Z. [www.intel.com](http://www.intel.com/) Order Number: 253667-028US

Учебно-методическое издание

# Гречишников Виктор Александрович Логинова Людмила Николаевна

**МАШИННО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЯЗЫК АССЕМБЛЕРА (ДЛЯ INTEL P6)**

Методические указания к лабораторным работам

по дисциплинам «Машинно-ориентированные языки программирования» и

«Микропроцессорные системы в электроснабжении»

|  |  |  |
| --- | --- | --- |
|  |  | Подписано в печать |
| Формат | Печ.л. |  |
| Заказ | Тираж | Цена |

127994, Москва, ул. Образцова, 15. Типография МИИТа.

1. Фрагменты, записанные в квадратных скобках, могут отсутствовать. [↑](#footnote-ref-1)