Программирование на языке ассемблера

Методические указания к лабораторным работам
по курсам «Языки программирования и методы трансляции»
(для 3 курса ФМПИ по направлению
01.03.02 Прикладная математика и информатика),
«Низкоуровневое программирование»
(для 3 курса ФМПИ по направлению
02.03.03 Математическое обеспечение и администрирование
информационных систем)

Составители: Лисицин Д.В., кафедра ТПИ, Петров Р.В., кафедра ПМт,
Сивак М.А., кафедра ТПИ

РАЗРАБОТКА ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА В СРЕДЕ MICROSOFT VISUAL STUDIO

Структура программы на языке Ассемблера. В данном лабораторном курсе мы будем рассматривать программирование на языке Ассемблера для 32-разрядной операционной системы Microsoft Windows (для 32-разрядных процессоров фирмы Intel, первым из которых был процессор 80386).

В процессоре есть несколько специальных «переменных» («быстрых» ячеек памяти), которые имеют зарезервированные имена и используются во многих командах. Эти «переменные» называются регистрами. Имеется восемь 32-разрядных регистров общего назначения EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, они могут участвовать в математических операциях или операциях обращения к памяти. Возможно обращение к младшим 16 бит регистров как в регистрам с именами AX, BX, CX, DX, SI, DI, SP, BP соответственно. В свою очередь каждый из первых четырех 16-разрядных регистров позволяет отдельно обращаться к своим старшим и младшим восьми байтам как к регистрам AH/AL, BH/BL, CH/CL и DH/DL.

Работа с памятью в языке Ассемблера достаточно специфична. В зависимости от используемой модели памяти могут выделяться различные сегменты (блоки) памяти программы, в каждом из которых может храниться код, данные, стек. Стек используется для оперативной работы с переменными и сохранения адреса возврата при вызове процедур и работает по принципу «первым вошел – последним вышел». Адрес вершины стека хранится в регистре ESP.

Работа с памятью особо проста для 32-разрядных приложений, когда используется плоская (несегментированная) модель памяти FLAT. Каждой программе может быть выделен блок памяти размером до 4 Гбайт, а сегменты, описываемые в программе, являются логическими. Адреса и указатели являются 32-битными.

Рассмотрим упрощенную структуру программы на языке Ассемблера (со знака «;» начинается комментарий).

```
.386   ; это ассемблерная директива, предписывающая Ассемблеру использовать
; набор операций для процессора 80386.
.MODEL FLAT, STDCALL ; Директива .MODEL определяет модель памяти
; программы. Параметр STDCALL говорит Ассемблеру о соглашениях при вызове
; процедур, принятом в 32-разрядной Windows: параметры помещаются в стек от
; последнего к первому, освобождает стек от помещенных в него параметров
; вызываемая процедура.
.DATA; директива определяет начало сегмента данных
; Далее помещается описание данных
.CODE; директива определяет начало сегмента кода
MAIN PROC ; директива PROC описывает процедуру, MAIN - имя процедуры
; Далее помещается код процедуры
MAIN ENDP; Директива ENDP завершает описание процедуры
END MAIN; Директива END завершает программу, MAIN - имя первой
; выполняемой процедуры
```

Ввод-вывод данных в программе на языке Ассемблера. Для демонстрации лабораторных работ необходимо обеспечить ввод и вывод данных в программе на языке Ассемблера. Наиболее просто сделать это в специальном типе приложений – «консольном приложении».

В Microsoft Visual Studio соответствующий шаблон доступен из пункта меню **Файл/Создать/Проект**, тип проекта **Visual C++**, шаблон **Консольное приложение Windows**.

По умолчанию Visual Studio не распознает файлы с кодом на языке Ассемблера. Для того чтобы включить поддержку языка Ассемблера необходимо настроить в проекте условия сборки, указав, какой программой необходимо компилировать файлы *.asm. Обычно требуется специально произвести такую настройку. Для этого необходимо в контекстном меню проекта выбрать пункт **Зависимости сборки/Настройки сборки…** и в открывшемся окне указать специальные правила сборки для файлов с расширением asm, установив галочку напротив правила **MASM (.targets, .props)**. Если это забыть сделать, то придется делать настройку для каждого файла в проекте (в его контекстном меню **Свойства**).

Консольное приложение работает в текстовом режиме, позволяя осуществлять ввод с клавиатуры и вывод на экран. Видимая часть консольных приложений называется окном консольного приложения. Для работы с консолью существует большое число функций интерфейса прикладного программирования API операционной системы Windows. Мы рассмотрим лишь некоторые из них.

Для своей работы некоторые консольные функции требуют получения стандартных дескрипторов (идентификаторов) ввода и вывода.

Для того чтобы получить дескрипторы, используется функция **GetStdHandle**, аргументом которой может являться следующая из трех констант, определяющая получаемый дескриптор: –10 для ввода, –11 для вывода, –12 для сообщения об ошибке. Функция возвращает дескриптор в регистре EAX.

Для вывода текстовой информации используется функция **WriteConsoleA**, параметры которой имеют следующий смысл:

1) дескриптор вывода, который получен при помощи функции GetStdHandle;

2) адрес буфера, в котором находится выводимый текст;

3) количество выводимых символов;

4) адрес переменной, в которую будет помещено количество действительно выведенных символов;

5) резервный параметр, должен быть равен нулю.

Заметим, что буфер, где находится выводимый текст, не обязательно должен заканчиваться нулем, поскольку для данной функции указывается количество выводимых символов.

Также отметим, что, поскольку информация выводится в консольном окне, кодировка всех строковых констант должна быть DOS-овской. Перекодировку можно осуществлять специальной функцией **CharToOem**. Первым параметром этой функции является адрес строки, которую следует перекодировать, а вторым параметром – адрес строки, куда следует поместить результат. Причем поместить результат можно и в строку, которую желаем перекодировать.

Для ввода с консоли используется функция **ReadConsoleA**. Значения параметров этой функции следующие:

1) дескриптор ввода, который получен при помощи функции GetStdHandle;

2) адрес буфера, в который будет помещена вводимая информация;

3) длина буфера;

4) адрес переменной, в которую будет помещено количество действительно введенных символов;

5) резервный параметр, должен быть равен нулю.

При выводе числовой (а также и символьной) информации может использоваться функция **wsprintfA**, которая преобразует заданный список переменных в символьную форму в соответствии с указанным списком форматов преобразования и помещает образованную строку в заданный буфер. Строка завершается нулем. Параметры функции имеют следующий смысл:

1) адрес буфера, в который будет помещена строка;

2) адрес строки со списком форматов;

3) список переменных.

Заметим, что каждая переменная из списка представляет собой отдельный параметр. Таким образом, функция имеет переменное число параметров. Возвращаемое через регистр EAX значение – длина строки.

Пример. Программа с вводом-выводом данных.

```
.386
.MODEL FLAT, STDCALL
; прототипы внешних функций (процедур) описываются директивой EXTERN,
; после знака @ указывается общая длина передаваемых параметров,
; после двоеточия указывается тип внешнего объекта – процедура
EXTERN  GetStdHandle@4: PROC
EXTERN  WriteConsoleA@20: PROC
EXTERN  CharToOemA@8: PROC
EXTERN  ReadConsoleA@20: PROC
EXTERN  ExitProcess@4: PROC; функция выхода из программы
EXTERN  lstrlenA@4: PROC; функция определения длины строки
EXTERN  wsprintfA: PROC; т.к. число параметров функции не фиксировано,
                         ; используется соглашение, согласно которому очищает стек
                         ; вызывающая процедура
.DATA; сегмент данных
STRN DB "Введите строку: ",13,10,0; выводимая строка, в конце добавлены
; управляющие символы: 13 – возврат каретки, 10 – переход на новую
; строку, 0 – конец строки; с использованием директивы DB
; резервируется массив байтов
FMT DB "Число %d", 0; строка со списком форматов для функции wsprintfA
DIN DD ?; дескриптор ввода; директива DD резервирует память объемом
; 32 бита (4 байта), знак «?» используется для неинициализированных данных
DOUT DD ?; дескриптор вывода
BUF  DB 200 dup (?); буфер для вводимых/выводимых строк длиной 200 байтов
```

```
LENS DD ?; переменная для количества выведенных символов
.CODE; сегмент кода
MAIN PROC; описание процедуры
; перекодируем строку STRN
MOV  EAX, OFFSET STRN;  командой MOV  значение второго операнда
; перемещается в первый, OFFSET – операция, возвращающая адрес
PUSH EAX; параметры функции помещаются в стек командой PUSH
PUSH EAX
CALL CharToOemA@8; вызов функции
; перекодируем строку FMT
MOV  EAX, OFFSET FMT
PUSH EAX
PUSH EAX
CALL CharToOemA@8; вызов функции
; получим дескриптор ввода
PUSH –10
CALL GetStdHandle@4
MOV DIN, EAX   ; переместить результат из регистра EAX
                                                    ; в ячейку памяти с именем DIN
; получим дескриптор вывода
PUSH –11
CALL GetStdHandle@4
MOV DOUT, EAX
; определим длину строки STRN
PUSH OFFSET STRN; в стек помещается адрес строки
CALL lstrlenA@4; длина в EAX
; вызов функции WriteConsoleA для вывода строки STRN
PUSH 0; в стек помещается 5-й параметр
PUSH OFFSET LENS; 4-й параметр
PUSH EAX; 3-й параметр
PUSH OFFSET STRN; 2-й параметр
PUSH DOUT; 1-й параметр
CALL WriteConsoleA@20
; ввод строки
PUSH 0; в стек помещается 5-й параметр
PUSH OFFSET LENS; 4-й параметр
PUSH 200; 3-й параметр
PUSH OFFSET BUF; 2-й параметр
PUSH DIN; 1-й параметр
CALL ReadConsoleA@20 ; обратите внимание: LENS больше числа введенных
; символов на два, дополнительно введенные символы: 13 – возврат каретки и
                                                    ; 10 – переход на новую строку
; вывод полученной строки
PUSH 0
```

```
PUSH OFFSET LENS
PUSH LENS; длина вводимой строки
PUSH OFFSET BUF
PUSH DOUT
CALL WriteConsoleA@20
; вывод числа 123 в буфер BUF
PUSH 123
PUSH OFFSET FMT
PUSH OFFSET BUF
CALL wsprintfA
ADD ESP, 12; очистка стека от параметров (изменение регистра ESP
                                                    ; на 3*4 = 12 байтов)
; вывод строки с числом 123
PUSH 0
PUSH OFFSET LENS
PUSH EAX
PUSH OFFSET BUF
PUSH DOUT
CALL WriteConsoleA@20
; небольшая задержка
MOV ECX,03FFFFFFH; помещение в регистр ECX – счетчик цикла –
                                                    ; большого значения
L1: LOOP L1; цикл без тела
; выход из программы
PUSH 0; параметр: код выхода
CALL ExitProcess@4
MAIN ENDP
END MAIN
```

Отладка программы на языке Ассемблера. Для отладки программы на языке Ассемблера используются те же возможности встроенного отладчика среды Visual Studio, что и при отладке программ на других языках, но с дополнениями, вызванными спецификой программирования на языке Ассемблера.

Специфика отладки программ на языке Ассемблера заключается в следующем:

– необходимо отслеживать состояние регистров;

– необходимо иметь возможность просматривать содержимое памяти по нужному адресу;

– необходимо просматривать содержимое стека.

Во время отладки становятся доступны несколько окон, позволяющих просматривать низкоуровневую информацию.

Окно **Регистры** содержит значения регистров процессора. Нужные группы регистров можно добавить в окно, воспользовавшись контекстным меню. При изменении значения регистра оно отображается красным цветом. В окне можно непосредственно изменять значения, содержащиеся в регистрах. Кроме того, со-

держимое регистров можно перетаскивать мышью в окна **Контрольные значения** и **Память**. Заметим, что флаги в этом окне имеют названия, отличные от общепринятых. В частности, флаг OF назван OV, DF – UP, SF – PL, ZF – ZR, CF – CY.

Окна **Память** позволяют пользователю просмотреть содержимое больших связных фрагментов памяти. Это может понадобиться при просмотре содержимого большого текстового буфера, строки или массива. Эти окна позволяют не только просматривать содержимое области памяти, но и редактировать его. При изменении значений они отображаются красным цветом. Панель инструментов данных окон позволяет выбрать начальный адрес отображаемого фрагмента и число столбцов в окне, а контекстное меню позволяет установить режим отображения информации как целого числа (длины 1, 2, 4 или 8 байт) или числа с плавающей запятой, а также текстовый формат, в который параллельно преобразуется данная информация. Целые числа могут выводиться в шестнадцатеричном формате, а также в формате целого числа со знаком или без знака. Содержимое буфера может вообще не преобразовываться в текстовый формат или выводиться в формате ASCII или Unicode.

В окне **Дизассемблированный код** выводится ассемблерный код исполняемого приложения. Это окно удобно для изучения кода, получаемого из программ на языках высокого уровня, таких как C/C++, а также для отладки приложений, написанных на нескольких языках (например, на языках C/C++ и Ассемблера).

Для просмотра стека необходимо отобразить на экране содержимое **сегмента стека**. Для этого в режиме отладки в строку **Адрес** окна **Память** записать название регистра ESP. В окне **Память** отобразится блок памяти, начиная с вершины стека. Однако, поскольку стек растет в направлении от старших адресов к младшим, для работы с ним удобно указать значение меньшее, чем находится в ESP. Для этого в строке Адрес можно записать ESP–n, где n число, как правило, кратное 4. Например, ESP–32.

Помимо названных окон, можно просматривать содержимое регистров и ячеек памяти в сегментах данных и стека через окна **Контрольные значения**. Однако, если для отображения значений нужно некоторое преобразование, то необходимо пользоваться, как правило, синтаксисом языка C++.

Например, чтобы отобразить **сегмент данных** удобно адрес первой ячейки памяти, зарезервированной в сегменте данных, указать в строке **Адрес** окна **Память**. Если эта ячейка памяти имеет имя Name, то в строке **Адрес** необходимо указать &Name. Так, в нашей программе это будет &STRN.

Упражнение. Начните пошаговую отладку программы из примера, нажав клавишу F10 или F11.

Добавьте в окно Контрольные значения следующие элементы (указаны через запятую):

&STRN, &STRN+8, EAX, (unsigned char*) EAX, (unsigned char*) (EAX+8),
ESP, *(unsigned long*) ESP, *(unsigned long*) (ESP+4),
(unsigned char*)*(unsigned long*) ESP, (unsigned char*)*(unsigned long*) (ESP+4),
(unsigned char*)*(unsigned long*)(ESP+4)+8.

Определите, исходя из синтаксиса языка C++, смысл этих элементов. Для колонки **Значения** установите режим отображения шестнадцатеричных значений через контекстное меню.

Откройте два окна **Память**, отобразите в них сегменты данных и стека. Установите режим отображения шестнадцатеричных значений и целых значений длиной 4 бита в сегменте стека, длиной 1 бит в сегменте данных.

Выполните пошагово три первые команды программы, отследите на каждом шаге изменение элементов в окне **Контрольные значения** и значений в сегменте стека, проанализируйте причины этих изменений.

Продолжите пошаговую отладку, отслеживая и анализируя изменение значений в сегментах данных и стека.

Откройте окно **Дизассемблированный код**, ознакомьтесь с его содержимым.

ЛАБОРАТОРНАЯ РАБОТА №1

ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА АССЕМБЛЕРА

1. Цель работы

Изучить и приобрести практические навыки работы с основными командами языка Ассемблера, функциями ввода-вывода, регистрами и символьными данными.

2. Содержание работы

1. Изучить основные команды языка Ассемблера, функции операционной системы, осуществляющие ввод-вывод, работу символьными данными.

2. По предложенному преподавателем варианту разработать программу на языке Ассемблера, решающую поставленную задачу:

1) ввод с клавиатуры числа в заданной системе счисления (система счисления А);

2) вывод введенного числа в десятичной системе счисления;

3) вычисление значения полинома вида $ax^2 + bx + c$ (в предположении, что результат вычислений не приводит к переполнению регистров);

4) вывод результата в заданной системе счисления (система счисления Б);

5) вывод результата в десятичной системе счисления.

Все промежуточные данные должны сохраняться в памяти. При выводе результата не использовать функцию wsprintfA. **Приглашение к вводу числа, а также вывод результатов на экран должны сопровождаться понятными пользователю текстовыми сообщениями.**

3. Отладить программу, убедиться в правильности ее работы на тестовых примерах.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. Методические указания

Основные команды языка Ассемблера. Рассмотрим кратко основные команды языка Ассемблера, которые могут понадобиться при выполнении задания лабораторной работы.

Команда **MOV** (move – переслать) пересылает байт, слово или двойное слово между регистром и ячейкой памяти или между двумя регистрами. Она может также пересылать константу в регистр или в ячейку памяти. Команда MOV имеет следующий формат (пересылка осуществляется из источника в приемник):

MOV приемник, источник

Для работы со стеком используются команды **PUSH** и **POP**:

PUSH источник

POP приемник

Команда PUSH помещает содержимое регистра, ячейки памяти или константу размером в слово или двойное слово на вершину стека. А команда POP снимает слово или двойное слово с вершины стека и помещает его в ячейку памяти или регистр.

Команда безусловного перехода **JMP** заставляет процессор извлечь новую команду не из следующей ячейки памяти, а из ячейки, связанной с операндом команды. Формат команды:

JMP имя

Команды **условной передачи управления** позволяют процессору «принять решение» о ходе исполнения программы в зависимости от определенных условий, например, нулевого значения регистра или единичного значения флага. Если такое условие выполнено, то процессор выполнит переход. В противном случае он продолжит исполнение со следующей команды программы. Формат команды:

Jx метка

Здесь x – модификатор, определяющий конкретную команду. Командам условной передачи управления могут предшествовать любые команды, изменяющие состояния флагов, но обычно они используются совместно с командой сравнения **CMP**:

CMP приемник, источник

Команда CMP вычитает операнд-источник из операнда-приемника и в зависимости от результата устанавливает или обнуляет флаги. Совместное использование команд отражено в таблице.

Таблица

Условие перехода	Следующая за CMP команда	
	для чисел без знака	для чисел со знаком
Приемник больше источника	JA	JG
Приемник равен источнику	JE	JE
Приемник не равен источнику	JNE	JNE
Приемник меньше источника	JB	JL
Приемник меньше источника или равен ему	JBE	JLE
Приемник больше источника или равен ему	JAE	JGE

Команды **управления циклами** обеспечивают условные передачи управления при организации циклов. Регистр ECX служит счетчиком числа повторений циклов. Основная команда этой группы LOOP (повторять цикл до конца счетчика) имеет формат

LOOP близкая_метка

Запись операнда близкая_метка подчеркивает, что метка перехода должна находиться не далее –128 или +127 байтов от команды. Команды управления циклами уменьшают содержимое регистра ECX на 1, а затем используют его новое значение для «принятия решения» о выполнении или не выполнении перехода на команду, отмеченную меткой. Команды LOOPE (LOOPZ) и LOOPNE (LOOPNZ) используют при решении о выполнении перехода также значение регистра ZF (1 и 0 соответственно).

Команды **ADD** (сложить) и **SUB** (вычесть) могут соответственно складывать и вычитать 8-, 16- и 32-битовые операнды. Форматы команд:

ADD приемник, источник
SUB приемник, источник

Команда **ADD** складывает содержимое операнда-источника и операнда-приемника и помещает результат в операнд-приемник. Команда SUB вычитает операнд-источник из операнда-приемника и возвращает результат в операнд-приемник.

Команда **INC** (increment – прирастить) добавляет 1 к содержимому регистра или ячейки памяти. Команда **DEC** (decrement – уменьшить) вычитает 1 из содержимого регистра или ячейки памяти. Форматы команд:

INC приемник
DEC приемник

Команда **MUL** умножает числа без знака, а **IMUL** – числа со знаком. Обе команды могут умножать байты, слова и двойные слова. Эти команды имеют формат

MUL источник
IMUL источник

где источник – регистр общего назначения или ячейка памяти размером в байт, слово или двойное слово. В качестве второго операнда команды используют содержимое регистра AL, AX или EAX при операциях над байтами, словами или двойными словами соответственно.

Произведение имеет *двойной размер* и возвращается следующим образом. Умножение байтов возвращает 16-битовое произведение в регистрах AH (старший байт) и AL (младший байт). Умножение слов возвращает 32-битовое произведение в регистрах DX (старшее слово) и AX (младшее слово). Умножение двойных слов возвращает 64-битовое произведение в регистрах EDX (старшее двойное слово) и EAX (младшее двойное слово).

Команда **DIV** выполняет деление чисел без знака, а команда **IDIV** выполняет деление чисел со знаком. Эти команды имеют формат

DIV источник
IDIV источник

где источник – делитель размером в байт, слово или двойное слово. Он находится в регистре общего назначения или в ячейке памяти. Делимое должно иметь *двойной размер*. Оно извлекается из регистров AH и AL (при делении на 8-битовое число), из регистров DX и AX (при делении на 16-битовое число), из регистров EDX и EAX (при делении на 32-битовое число).

Результаты возвращаются следующим образом. Если операнд-источник представляет собой байт, то частное возвращается в регистре AL, а остаток в регистре AH. Если операнд-источник представляет собой слово, то частное возвращается в регистре AX, а остаток – в регистре DX. Если операнд-источник представляет собой двойное слово, то частное возвращается в регистре EAX, а остаток – в регистре EDX.

Преобразования строк и чисел. Ввод информации с клавиатуры и вывод ее на экран в консольных приложениях осуществляется только в символьном виде. При этом каждый символ кодируется одним байтом согласно стандарту ASCII/ANSI. Таким образом, чтобы ввести число, необходимо сначала ввести строку, а затем преобразовать ее в двоичное число.

Пример. Формирование числа из строки символов десятичной системы счисления. Пусть s – исходная строка, ax – результирующее число.

ax = 0;
for (i=0; i<длина_строки(s); i++) ax = ax*10 + s[i] – '0';

Необходимость вычитания из очередного символа строки s[i] символа '0' связана с тем, что цифры в таблице ASCII имеют коды от 30H для символа '0' до 39H для символа '9'. Таким образом, получаем нужное число путем вычитания из кода текущего символа кода числа 0.

Рассмотрим реализацию этого алгоритма на языке Ассемблера для случая, когда число накапливается в регистре AX.

MOV DI, 10; основание системы счисления
MOV ECX, LENS; счетчик цикла (строка имеет длину LENS)
MOV ESI, OFFSET BUF; начало строки хранится в переменной BUF
XOR BX, BX; обнулить регистр BX командой XOR,

```
                ; выполняющей побитно операцию «исключающее или»
    XOR AX, AX; обнулить регистр AX
    CONVERT:    ; метка начала тела цикла
        MOV BL, [ESI]; поместить символ из введенной строки в регистр
                     ; BL, используя косвенную адресацию
        SUB BL, '0'; вычесть из введенного символа код нуля
        MUL DI; умножить значение AX на 10, результат – в AX
        ADD AX, BX; добавить к полученному в AX числу новую цифру
        INC ESI; перейти на следующий символ строки
    LOOP CONVERT; перейти на следующую итерацию цикла
```

Заметим, что данная реализация является схематичной, поскольку, например, в ней не проверяется корректность вводимых данных.

Аналогично, для вывода числа нам нужно преобразовать его в строку. В этом случае необходимо в цикле делить данное число на 10 (или основание иной системы счисления) до тех пор, пока делимое не окажется равным нулю, и записывать остаток от деления в строку. В результате получится строка, в которой выводимое число записано в обратном порядке. После инверсии строки можно вывести число на экран, предварительно переведя каждый символ в код ASCII. Для инверсии строки удобно использовать стек. Можно изначально помещать остатки в стек с последующим формированием «прямой» строки.

4. Варианты заданий

Вариант	Система счисления А	Система счисления Б	Коэффициенты полинома		
			a	b	c
1	2	8	5	18	-1
2	8	16	3	-12	3
3	2	16	-21	8	4
4	8	8	6	-3	15
5	16	8	4	19	-2
6	2	2	-1	8	10
7	8	2	11	-2	1
8	16	2	8	18	-5
9	16	16	-14	9	4
10	2	8	3	-15	1
11	16	2	23	8	-7
12	8	16	-3	29	6

5. Контрольные вопросы

1. Функция GetStdHandle: назначение, параметры, примеры использования.
2. Функция WriteConsoleA: назначение, параметры, примеры использования.
3. Функция CharToOem: назначение, параметры, примеры использования.

4. Функция ReadConsoleA: назначение, параметры, примеры использования.

5. Функция wsprintfA: назначение, параметры, примеры использования.

6. Команда mov: назначение, операнды, примеры использования.

7. Стек и команды работы со стеком push, pop: назначение, операнды, примеры использования.

8. Команды сохранения и извлечения флагов и регистров из стека.

9. Команды сложения.

10. Команды вычитания.

11. Команды деления.

12. Команды умножения.

13. Директивы определения данных.

14. Регистровая, непосредственная и прямая адресация.

15. Косвенная регистровая адресация и адресация по базе.

16. Прямая адресация с индексированием и адресация по базе с индексированием.

ЛАБОРАТОРНАЯ РАБОТА № 2

ЦЕПОЧЕЧНЫЕ КОМАНДЫ

1. Цель работы

Изучить и приобрести практические навыки работы с цепочечными командами (командами обработки строк символов).

2.Содержание работы

1. Изучить цепочечные команды.

2. По предложенному преподавателем варианту разработать программу на языке Ассемблера, решающую поставленную задачу с использованием цепочечных команд.

3. Отладить программу, убедиться в правильности ее работы на тестовых примерах.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, алгоритм, текст разработанной программы и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. Методические указания

Общая характеристика цепочечных команд. Система команд процессора имеет группу команд, позволяющих производить действия над блоками элементов

(цепочками). Логически цепочки представляют собой последовательности элементов с любыми значениями (символы, числа и прочее), хранящихся в соседних ячейках памяти в виде двоичных кодов. Единственное ограничение состоит в том, что размеры элементов в цепочках фиксированы значениями байт, слово или двойное слово.

Команды обработки строк обеспечивают возможность выполнения ряда операций-примитивов, обрабатывающих цепочки поэлементно.

Каждая операция-примитив реализуется в процессоре тремя командами, в свою очередь, каждая из этих команд работает с соответствующим размером элемента – байтом, словом или двойным словом. Кроме того, имеется одна команда с явным указанием операндов, служащих только для определения типов операндов. После того как выяснен тип элементов цепочек по их описанию в памяти, генерируется одна из трех машинных команд.

Все цепочечные команды обрабатывают только один (текущий) элемент цепочки и автоматически передвигаются к следующему элементу.

Адреса текущих обрабатываемых элементов всегда находятся в регистрах ESI для строки, участвующей в команде в качестве источника, и EDI для строки, участвующей в команде в качестве приемника.

Переход к следующему элементу производится увеличением или уменьшением содержимого регистров ESI, EDI.

Возможны два направления обработки цепочки:

– от начала цепочки к ее концу, то есть в направлении возрастания адресов;

– от конца цепочки к началу, то есть в направлении убывания адресов.

Направление определяется значением флага направления DF в регистре EFLAGS/FLAGS:

– если DF = 0, то значения индексных регистров ESI, EDI будут автоматически увеличиваться цепочечными командами, то есть обработка будет осуществляться в направлении возрастания адресов;

– если DF = 1, то значения индексных регистров ESI, EDI будут автоматически уменьшаться цепочечными командами, то есть обработка будет идти в направлении убывания адресов.

Состоянием флага DF можно управлять с помощью двух команд, не имеющих операндов:

– **CLD** – очистить флаг направления (команда сбрасывает флаг направления DF в 0);

– **STD** – установить флаг направления (команда устанавливает флаг направления DF в 1).

Количество байтов, на которые изменяется содержимое регистров ESI, EDI определяется типом элементов строки или кодом цепочечной команды: на 1 при обработке байтов, на 2 при обработке слов и на 4 при обработке двойных слов.

Поскольку адреса текущих обрабатываемых элементов извлекаются командами из регистров ESI и EDI, перед выполнением команд необходимо нужные адреса в эти регистры поместить. Обычно перед началом обработки строк в указанные регистры заносятся адреса первых обрабатываемых элементов. Для этого мо-

гут использоваться команда LEA или уже использовавшаяся нами ранее операция OFFSET.

Команда LEA (загрузить исполнительный адрес) пересылает смещение ячейки памяти в 32-битовый регистр. Она имеет формат

LEA регистр, память

Префиксы повторения. Логически к командам обработки строк нужно отнести и так называемые префиксы повторения. Префиксы повторения имеют мнемонические обозначения REP, REPE (REPZ), REPNE (REPNZ).

Префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле. Число повторений заносится в регистр ECX. После очередного выполнения команды значение в регистре CX уменьшается на единицу. Решение об очередном выполнении цепочечной команды принимается по состоянию регистра ECX и, возможно, по флагу нуля ZF.

Префикс повторения REP (REPeat) заставляет команды выполняться, пока содержимое в ECX не станет равным 0.

Префиксы повторения REPE (REPeat while Equal) и REPZ (REPeat while Zero) являются синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое ECX не равно 0 и флаг ZF равен 1. Как только одно из этих условий нарушается, управление передается следующей команде программы. Префиксы REPNE и REPNZ, также являющиеся синонимами, заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое ECX не равно 0 и флаг ZF равен 0. При нарушении одного из этих условий работа команды прекращается.

Пересылка цепочек: команда MOVS. Синтаксис команды MOVS:

MOVS адрес_приемника, адрес_источника

Команда копирует элемент цепочки-источника (длиной в байт, слово или двойное слово), который располагается по адресу, соответствующему содержимому регистра ESI, в элемент цепочки-приемника, который располагается по адресу, соответствующему содержимому регистра EDI. Команды MOVSB, MOVSW и MOVSD выполняют аналогичные действия. При этом первая команда оперирует элементами длиной в байт, вторая – элементами длиной в слово, третья – элементами длиной в двойное слово.

Последовательность действий, которые нужно выполнить в программе для того, чтобы переслать цепочку элементов из одной области памяти в другую с помощью команды MOVS следующая:

1. Установить значение флага DF в зависимости от того, в каком направлении будут обрабатываться элементы цепочки – в направлении возрастания или убывания адресов.

2. Загрузить адреса первых обрабатываемых элементов цепочек в регистры ESI и EDI.

3. Загрузить в регистр ECX количество обрабатываемых элементов.

4. Записать команду MOVS с префиксом REP.

Пример. Пусть необходимо скопировать 100 элементов из строки SOURCE в строку DEST, обе строки описаны сегменте данных как массивы байтов.

```
CLD
LEA     ESI, SOURCE
LEA     EDI, DEST
MOV     ECX, 100
REP     MOVSB
```

Сравнение цепочек: команда CMPS. Команды, реализующие операцию-примитив сравнения цепочек, производят сравнение элементов цепочки-источника с элементами цепочки-приемника. Синтаксис команды CMPS:

```
CMPS    адрес_приемника, адрес_источника
```

Алгоритм работы команды CMPS заключается в вычитании элемента цепочки-приемника, который располагается по адресу, соответствующему содержимому регистра EDI, из элемента цепочки-источника, который располагается по адресу, соответствующему содержимому регистра ESI. Команда производит вычитание элементов, не записывая при этом результата, и устанавливает флаги ZF, SF и OF. Так, если флаг нуля ZF равен 1, то элементы оказались равны, а если ZF равен 0, то, наоборот, не равны.

Чтобы заставить команду CMPS выполняться несколько раз, то есть произвести последовательное сравнение элементов цепочек, необходимо перед командой CMPS определить префикс повторения. С командой CMPS удобно использовать префиксы повторения REPE/REPZ или REPNE/REPNZ для поиска первой пары несовпадающих или, наоборот, совпадающих элементов. Поскольку при выполнении команды CMPS значения регистров ESI и EDI изменяются, по завершении цикла они указывают не на искомые элементы в цепочке, а на следующие, которые должны были бы сравниваться.

Ассемблер преобразует команду CMPS в команду CMPSB (при сравнении байтов), в команду CMPSW (при сравнении слов) или в команду CMPSD (при сравнении двойных слов) в зависимости от описания операндов.

Сканирование цепочек: команда SCAS. Команды, реализующие операцию-примитив сканирования цепочек, производят поиск некоторого значения в цепочке. Искомое значение предварительно должно быть помещено в один из регистров AL, AX или EAX (в зависимости от размера элемента цепочки).

Синтаксис команды SCAS:

```
SCAS    адрес_приемника
```

Принцип поиска здесь тот же, что и в команде сравнения CMPS, то есть вычитание элемента цепочки-приемника, который располагается по адресу, соответствующему содержимому регистра EDI, из содержимого регистра AL/AX/EAX. В зависимости от результатов вычитания производится установка флагов, при этом сами операнды не изменяются.

Так же как и в случае команды CMPS, с командой SCAS удобно использовать префиксы REPE/REPZ или REPNE/REPNZ.

Ассемблер преобразует команду SCAS в команду SCASB (при поиске байта), в команду SCASW (при поиске слова) или в команду SCASD (при поиске двойного слова).

Загрузка элемента цепочки в регистр: команда LODS. Операция-примитив загрузки элемента цепочки в регистр позволяет скопировать элемент цепочки в регистр AL, AX, EAX (в зависимости от размера элемента цепочки).

Синтаксис команды LODS:

LODS адрес_источника

Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому регистра ESI, и поместить его в регистр AL/AX/EAX.

Ассемблер преобразует команду LODS в команду LODSB (при работе с цепочкой байтов), в команду LODSW (при работе с цепочкой слов) или в команду LODSD (при работе с цепочкой двойных слов).

Сохранение содержимого регистра в цепочке: команда STOS. Операция-примитив сохранения содержимого регистра в цепочке позволяет произвести действие, обратное действию команды LODS, то есть перенести значение из регистра в элемент цепочки. Синтаксис команды STOS:

STOS адрес_приемника

Команда пересылает элемент из регистра AL/AX/EAX в элемент цепочки по адресу, соответствующему содержимому регистра EDI.

Ассемблер преобразует команду STOS в команду STOSB (при работе с цепочкой байтов), в команду STOSW (при работе с цепочкой слов) или в команду STOSD (при работе с цепочкой двойных слов).

4. Варианты заданий

1. Написать программу формирования сжатой строки символов. Сжатие заключается в удалении повторного вхождения символов из исходной строки при просмотре ее слева направо.
2. Написать программу формирования сжатой строки символов. Сжатие заключается в удалении повторного вхождения символов из исходной строки при просмотре ее справа налево.
3. Написать программу выделения из исходной строки подстроки символов заданной длины с указанного номера позиции по направлению к началу строки.
4. Написать программу, выделяющую из исходной строки подстроку заданной длины с указанного номера позиции и инвертирующую эту подстроку.
5. Написать программу, выделяющую из исходной строки подстроку заданной длины с указанного номера позиции и приводящую символы этой подстроки к верхнему регистру.
6. Написать программу, выделяющую из исходной строки подстроку заданной длины с указанного номера позиции и приводящую символы этой подстроки к нижнему регистру.
7. Написать программу, которая по исходной строке формирует палиндром.

8. Написать программу, инвертирующую регистр символов в исходной строке (заглавные буквы переводятся в строчные и наоборот).

9. Написать программу, заменяющую в исходной строке все вхождения первого введенного символа на второй.

10. Написать программу, вставляющую в исходную строку пробел через каждые n символов (n вводится пользователем).

11. Написать программу, дублирующую каждый символ в исходной строке (например, из строки abcd должна получиться строка aabbccdd).

12. Написать программу, удаляющую точки в исходной строке. Если последний символ является точкой, он не удаляется.

5. Контрольные вопросы

1. Общая характеристика цепочечных команд.
2. Префиксы повторения.
3. Пересылка цепочек.
4. Сравнение цепочек.
5. Сканирование цепочек.
6. Загрузка элемента цепочки в регистр.
7. Сохранение содержимого регистра в цепочке.

ЛАБОРАТОРНАЯ РАБОТА № 3

ИНТЕРФЕЙС С ЯЗЫКОМ С++

1. Цель работы

Изучить и приобрести практические навыки использования процедур на языке Ассемблера и их интерфейса с языком С++.

2. Содержание работы

1. Изучить способы вызова процедуры на языке Ассемблера, возврата из процедуры.

2. Изучить условия взаимодействия функции на языке С++ с процедурой на языке Ассемблера. Изучить правила передачи управления в процедуру и обратно. Изучить способы обмена данными между вызывающей функцией на языке С++ и процедурой на языке Ассемблера.

3. Написать процедуру на языке Ассемблера, реализующую функцию заданного варианта **с использованием цепочечных команд**. Написать вызывающую функцию на языке С++, осуществляющую ввод исходных данных и вывод результатов.

4. Отладить программу, убедиться в правильности ее работы на тестовых примерах.

5. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, алгоритм, текст разработанной программы и результаты тестирования.

6. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. Методические указания

Процедуры. Процедура в программе на языке Ассемблера определяется директивой PROC. Определение процедуры выглядит следующим образом:

```
имя_процедуры        PROC
; тело процедуры
RET [n]
имя_процедуры        ENDP
```

Операнд имя_процедуры определяет метку начала процедуры. Директива ENDP определяет конец процедуры и имеет имя, аналогичное имени в директиве PROC. Команда RET завершает выполнение процедуры.

Команды CALL (вызвать процедуру) и RET (возвратиться из процедуры) обеспечивают исполнение процедур.

Команда **CALL** имеет формат

```
CALL    имя
```

где операндом является имя вызываемой процедуры (точнее, метка ее начала).

Команда CALL осуществляет функции запоминания адреса возврата и передачи управления процедуре. Она помещает в стек адрес возврата (32-битный адрес следующей на ней команды). После сохранения адреса возврата команда CALL загружает адрес начала процедуры в указатель команд EIP.

Команда **RET** заставляет процессор возвратиться из процедуры в место ее вызова. Команда RET обязательно должна быть последней командой процедуры, исполняемой процессором.

Команда RET извлекает из стека адрес возврата и загружает его в указатель команд EIP. Если указан необязательный операнд n, то между извлечением из стека адреса возврата и передачей управления в точку возврата из стека удаляется n байтов.

Основы взаимодействия языков C++ и Ассемблера. В среде Microsoft Visual Studio возможна разработка программ, разные модули которых написаны на разных языках: C++ и Ассемблера. Для создания такой программы необходимо создать проект, в который включить необходимые модули, написанные на языке C++ и языке Ассемблера.

Рассмотрим случай, когда функция на языке C++ вызывает процедуру на языке Ассемблера.

Важное требование к взаимодействию состоит в том, что процедура на языке Ассемблера и функция на языке C++ придерживались одинаковых соглашений о вызовах. Функция и процедура должны придерживаться общих соглашений о пе-

редаче данных процедуре и о возвращении данных из процедуры в функцию. Также функции и процедуре нужно придерживаться таких соглашений об использовании машинных регистров, чтобы процедура не уничтожала регистровых значений функции.

Чтобы функция на языке C++ могла вызвать процедуру на языке Ассемблера и соответствующим образом получить управление обратно, должны быть выполнены следующие шаги. Функция сохраняет в стеке адрес команды, с которой будет продолжено ее исполнение после завершения вызова процедуры. Затем функция передает управление процедуре. После завершения процедура возвращает управление по адресу, сохраненному функцией в стеке.

Передача данных процедуре. Существуют два способа обмена данными между функцией на языке C++ и процедурой на языке Ассемблера: использование глобальных данных и использование аргументов.

Использование глобальных данных. Процедура на языке Ассемблера может иметь доступ к глобальным данным, определенным в модуле на языке C++ с использованием внешнего класса хранения данных. В процедуре необходимо использовать для ссылки на внешние объекты директиву EXTERN.

Пример. Процедура изменяет значение внешнего объекта VAL.

Содержание модуля TEST1.CPP.

```
extern "C" void COPY( ); // объявить внешнюю функцию (процедуру)
extern "C" int VAL = 0; // определить внешний объект данных,
// инициализация обязательна
void main( ) {
COPY( );
printf("VAL = %d\n", VAL);
}
```

Содержание модуля COPY.ASM.

```
.386
.MODEL FLAT;
EXTERN _VAL: DWORD; имя _VAL объявлено как глобальный объект
                    ; длиной в двойное слово
.CODE
_COPY       PROC
MOV _VAL, 4; модифицируется глобальная переменная
RET; возвращение в вызывающую функцию
_COPY       ENDP
END
```

Замечания.

1. Компилятор C++ добавляет знак подчеркивания ко всем глобальным идентификаторам, поэтому в модуле на языке Ассемблера используются переменная _VAL и процедура _COPY.

2. Хотя основной модуль написан на языке C++, в нем используется связь с языком Ассемблера в стиле языка Си, для чего в модуле на языке C++ в объявле-

ниях процедуры и используемой ею глобальной переменной используется описатель "C".

Использование аргументов функции. В языке C++ используется несколько разных соглашений о вызовах, определяющих, в частности, способ передачи аргументов. По умолчанию в языке C++ используется соглашение, называемое **cdecl**.

Согласно соглашению **cdecl** аргументы передаются по значению, т.е. вызываемая процедура получает копию каждого аргумента. Аргументы помещаются в стек в порядке, обратном тому, в котором они указаны при вызове процедуры. По окончании выполнения процедуры вызывающая функция очищает стек от аргументов.

Пример. Если A1, A2, A3 – переменные типа int, то вызов процедуры FUNC(Al, A2, A3) преобразуется компилятором C++ в последовательность команд

```
PUSH A3
PUSH A2
PUSH A1
CALL _FUNC
ADD ESP, 12
```

Ячейка с адресом возврата имеет адрес [ESP]. Копии аргументов имеют адреса [ESP]+4, [ESP]+8, [ESP]+12.

Общепринятым считается обращение к аргументам в стеке через регистр указателя базы EBP. Поэтому одной из первых команд любой процедуры, которой необходимо адресоваться к аргументам, является помещение в регистр EBP значения регистра ESP. Перед этим необходимо сохранить значение EBP в стеке, а затем воспользоваться регистром EBP в процедуре; непосредственно перед возвратом в вызывающую функцию нужно восстановить исходное значение EBP.

Пример. Рассмотрим вызов процедуры, которая получает три целых аргумента, а возвращает полученный результат через внешнюю переменную VAL.

Содержание модуля TEST2.CPP.

```
extern "C" void FUNC(int A1, int A2, int A3);
extern "C" int VAL = 0;
void main( ){
int x = 10;
FUNC(x, 20, 20+5);
printf("VAL = %d\n", VAL);
}
```

Аргументами вызова могут быть идентификаторы, константы или выражения. Каждое выражение вычисляется, а его значение помещается в стек командой PUSH.

Другим соглашением о вызовах является **stdcall**, согласно которому аргументы помещаются в стек в обратном порядке, но освобождает стек от помещенных в него аргументов вызываемая процедура.

В этом случае описание внешней процедуры в модуле на C++ выглядит следующим образом:

```
extern "C" void __stdcall FUNC(int A1, int A2, int A3); // используется двойное
```

```
                                                    // подчеркивание
```

В модуле на языке Ассемблера процедура должна иметь имя _FUNC@12 (после знака @ указывается общая длина передаваемых аргументов), а команда RET должна иметь аргумент, равный этой длине, т.е. 12.

Еще одно соглашение – **fastcall** – использует самый быстрый способ передачи аргументов – через регистры. Согласно ему первый и второй аргументы передаются через регистры ECX и EDX соответственно. Остальные аргументы передаются через стек в обратном порядке. Освобождает стек от помещенных в него аргументов вызываемая процедура.

Описание внешней процедуры в модуле на C++ здесь выглядит следующим образом:

```
extern "C" void __fastcall FUNC(int A1, int A2, int A3); // используется
                                                    // двойное подчеркивание
```

В модуле на языке Ассемблера процедура должна иметь имя @FUNC@12, команда RET должна иметь аргумент 4 (через стек передан только один аргумент).

Возвращение значений. Наряду с возвращением значений через глобальные объекты процедура на языке Ассемблера может возвращать их через аргументы вызова или как значение процедуры.

Чтобы вызывающая функция могла воспринять возвращаемый результат как значение имени функции (процедуры), нужно следующим образом модифицировать модуль TEST2.CPP.

Содержание модуля TEST3.CPP.

```
extern "C" int FUNC(int A1, int A2, int A3);
void main( ){
int val, x = 10;
val = FUNC(x, 20, 20+5);
printf("val = %d\n , val);
}
```

Вызывающая функция и процедура должны придерживаться определенных соглашений относительно возвращения значения через имя процедуры. Обычно в компиляторах языка C++ предполагается, что это значение запомнено в одном или нескольких регистрах.

Для рассматриваемого нами компилятора возвращаемое значение длиной 8 бит помещается в регистр AL, длиной 16 бит – в регистр AX, длиной 32 бита – в регистр EAX, длиной 64 бита – в пару регистров EDX:EAX. Структуры большего размера размещаются в статической памяти, а ссылки на них передаются через регистр EAX.

Если процедура должна возвращать несколько значений, то она может использовать для этой цели свои аргументы. В этом случае процедуре передается адрес аргумента, т.е. осуществляется передача аргумента по адресу (по ссылке).

Вызов функций на языке C++ из процедур на языке Ассемблера. Чаще всего приходится вызывать процедуры на языке Ассемблера из функций на C++. Однако можно вызывать и функции на языке C++ из процедур на языке Ассембле-

ра. Для этого надо объявить имя функции на языке C++ в модуле на языке Ассемблера, используя директиву EXTERN.

Если имя функции на языке C++ – CFUNC, то ее объявление при использовании соглашения **cdecl** будет иметь вид

EXTERN _CFUNC: PROC

Чтобы вызвать функцию на языке C++, каждый ее аргумент нужно поместить в стек, начиная с последнего аргумента, а затем вызвать функцию с помощью команды CALL. После возвращения из функции на языке C++ процедура на языке Ассемблера должна очистить стек, удалив из него все ранее помещенные аргументы. Для этого можно увеличить содержимое указателя стека ESP на целое значение, равное числу байтов, ранее помещенных в стек.

Если данные определены внутри модуля на языке Ассемблера, то их можно сделать доступными в модуле на языке C++, объявив общедоступными с помощью директивы PUBLIC в модуле на языке Ассемблера и внешними с помощью описания extern "C" в модуле на языке C++.

Пример. Вызов функции на языке C++ из процедуры на языке Ассемблера.

Содержание модуля TEST4.CPP.

```
extern "C" int SUM; // переменная описана в модуле на языке Ассемблера
extern "C" void CFUNC(int a, int b){ // используется соглашение cdecl
        SUM = a+b;
}
```

Содержание модуля ASMPRG.ASM.

```
.386
.MODEL FLAT
PUBLIC _SUM
EXTERN _CFUNC: PROC
EXTERN  _ExitProcess@4: PROC; используется соглашение stdcall
.DATA
_SUM DD ?
.CODE
ASMPRG    PROC
PUSH 3
PUSH 5
CALL _CFUNC
ADD ESP, 8
PUSH 0
CALL _ExitProcess@4
ASMPRG    ENDP
END ASMPRG
```

4. Варианты заданий

Варианты заданий см. в лабораторной работе 2. Для передачи данных процедуре на языке Ассемблера использовать аргументы функции. Для вызова процедуры использовать следующие соглашения о вызовах:

– варианты 1 – 4: stdcall;
– варианты 5 – 8: cdecl;
– варианты 9 – 12: fastcall.

5. Контрольные вопросы

1. Определение процедуры.
2. Команда вызова процедуры CALL.
3. Команда возврата из процедуры.
4. Косвенный вызов процедуры.
5. Основы взаимодействия языков C++ и Ассемблера.
6. Использование глобальных данных для передачи данных процедуре на языке Ассемблера.
7. Использование аргументов функции для передачи данных процедуре на языке Ассемблера.
8. Возвращение значений из процедуры на языке Ассемблера.
9. Вызов функций на языке C++ из процедур на языке Ассемблера.
10. Использование локальных данных.
11. Использование вставок на языке Ассемблера в программе на языке C++.

ЛАБОРАТОРНАЯ РАБОТА № 4

ПРОГРАММИРОВАНИЕ СОПРОЦЕССОРА

1. Цель работы

Изучить архитектуру и средства программирования сопроцессора (модуля операций с плавающей точкой) на языке ассемблера и приобрести практические навыки работы с ними.

2. Содержание работы

1. Изучить архитектуру и средства программирования сопроцессора на языке ассемблера.

2. Написать программу, реализующую вычисление функции в точке (вводится пользователем) в соответствии с заданным вариантом.

Программа должна состоять из модулей на C++ и ассемблере, причем в модуле на C++ осуществляется ввод-вывод, а все вычисления – в модуле на ассемблере.

3. Отладить программу, убедиться в правильности ее работы на тестовых примерах.

4. Оформить отчет по лабораторной работе. Отчет должен содержать постановку задачи, график функции, текст разработанной программы (на этапах вычислений в комментариях указывать состояние стека сопроцессора) и результаты тестирования.

5. Защитить лабораторную работу, ответив на вопросы преподавателя.

3. Методические указания

Форматы данных. Сопроцессор специально разрабатывался для вычислений с плавающей точкой. Он может работать с вещественными числами в трех форматах: одинарной точности (32 бита), двойной точности (64 бита), расширенной точности (80 битов); а также с целыми числами размера 16, 32 и 64 бита и упакованными двоично-десятичными числами. В самом сопроцессоре числа имеют одинаковое внутреннее представление – формат расширенной точности.

Вещественное число определяется директивами DD (или REAL4), DQ (или REAL8), DT (или REAL10) в зависимости от формата. В записи десятичной константы при этом обязательным является наличие десятичной точки, даже если она не имеет дробной части. Другой способ задания вещественного числа – экспоненциальная форма с использованием символа «E». Примеры:

```
DD 45.56
DD 0.4556E2
```

Архитектура сопроцессора. С точки зрения программиста, сопроцессор представляет собой совокупность регистров, каждый из которых имеет свое функциональное назначение.

Восемь регистров **R0...R7** составляют стек сопроцессора. Размерность каждого регистра составляет 80 битов – для хранения вещественных чисел расширенной точности. Команды сопроцессора не оперируют физическими номерами регистров стека R0...R7. Вместо этого они используют логические номера этих регистров **ST(0)...ST(7)**. Вершиной стека является регистр ST(0), к которому можно обращаться также как к ST.

Регистр состояния сопроцессора **SWR** отражает информацию о текущем состоянии сопроцессора и содержит поля, позволяющие определить, какой регистр является текущей вершиной стека сопроцессора, какие исключения возникли после выполнения последней команды, каковы особенности выполнения последней команды (некий аналог регистра флагов основного процессора) и т. д.

Регистр управления сопроцессора **CWR** управляет режимами работы сопроцессора; с помощью полей в этом регистре можно регулировать точность выполнения численных вычислений, управлять округлением, маскировать исключения.

Регистр тегов **TWR** используется для контроля состояния каждого из регистров R0...R7.

Использование отладчика. Встроенный отладчик среды Visual Studio предоставляет широкие возможности для отладки программ, использующих сопроцессор.

Для наблюдения за состоянием регистров сопроцессора можно в окно **Регистры** добавить группу «Floating Point», воспользовавшись контекстным меню. Регистры стека отображаются здесь в экспоненциальной форме (отмечаются и специальные численные значения) и называются ST0, ST1 и т.д. Регистр состояния называется STAT, регистр управления – CTRL, регистр тегов – TAGS, все эти регистры отображаются в шестнадцатеричной системе счисления, что, впрочем, не очень удобно для анализа их отдельных битов.

Для просмотра вещественных чисел, хранящихся в памяти, в окне **Память** можно в контекстном меню установить режим отображения чисел с плавающей запятой одинарной или двойной точности. Для чисел расширенной точности можно просматривать только их коды, установив режим отображения однобайтовых целых чисел.

Через окна **Контрольные значения** можно просматривать содержимое ячеек памяти и регистров стека сопроцессора (как переменных с именами ST0, ST1 и т.д.). Однако, если вещественные переменные описаны директивами DD и DQ, то они отображаются как целочисленные и нужно производить преобразование типов в соответствии с синтаксисом языка C++. Для вещественных переменных, описанных директивами REAL4, REAL8, REAL10 и DT, никаких преобразований не требуется.

Инициализация сопроцессора. Первый фрагмент программы с командами сопроцессора должен начинаться с команды инициализации сопроцессора **FINIT**, приводящей сопроцессор в некоторое начальное состояние. Она инициализирует управляющие регистры сопроцессора определенными значениями.

Рассмотрим кратко основные команды сопроцессора для работы с вещественными данными, которые могут понадобиться при выполнении задания лабораторной работы. Заметим, что у сопроцессора имеются также команды для работы с целочисленными и двоично-десятичными данными.

Команды передачи данных. Данные команды предназначены для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти.

Загрузка вещественного числа из области памяти или из другого регистра сопроцессора на вершину стека сопроцессора:

FLD источник

Сохранение вещественного числа из вершины стека сопроцессора в память или пустой регистр стека:

FST приемник

FSTP приемник

В команде FST можно указывать переменные в памяти размером только в 32 и 64 бита. В команде FSTP дополнительно осуществляется выталкивание вещественного числа из стека после его сохранения в памяти, в этой команде можно указывать переменные в памяти размером в 32, 64 и 80 битов.

Примеры.

```
FLD MEM_DOUBLE
FLD ST(4)  ; Загрузка значения из ST(4) в ST(0).
FLD ST     ; Дублирование вершины стека ST(1) = ST(0).
FST DWORD PTR [EBX]
FSTP ST ; удаление числа с вершины стека
```

Команда **обмена** вершины регистрового стека ST(0) с любым другим регистром стека сопроцессора ST(i):

FXCH ST(i)

Команды загрузки констант:

FLDZ – загрузка нуля;
FLD1 – загрузка единицы;
FLDPI – загрузка числа π;
FLDL2T – загрузка двоичного логарифма десяти;
FLDL2E – загрузка двоичного логарифма экспоненты (числа *e*);
FLDLG2 – загрузка десятичного логарифма двойки;
FLDLN2 – загрузка натурального логарифма двойки.

Команды сравнения данных. Команды сравнивают значение числа в вершине стека ST(0) и источника – 32- или 64-битной вещественной переменной или регистра ST(i) (если операнд не указан, то подразумевается ST(1)):

FCOM [источник]
FCOMP [источник]
FCOMPP

Последним действием второй команды является выталкивание значения из ST(0). Последним действием третьей команды является выталкивание из стека значений регистров ST(0) и ST(1).

Команда

FTST

не имеет операндов и сравнивает значения в ST(0) с нулем.

В результате работы команд сравнения в регистре состояния устанавливаются значения битов кода условия C3, C2, C0 из регистра SWR, указанные в таблице.

Условие	C3 (ZF)	C2 (PF)	C0 (CF)
ST(0) > операнда	0	0	0
ST(0) < операнда	0	0	1
ST(0) = операнду	1	0	0
операнды неупорядочены	1	1	1

Для того чтобы получить возможность реагировать на результаты сравнения командами условного перехода основного процессора (они реагируют на флаги в EFLAGS), нужно записать сформированные биты условия C3, C2, C0 в регистр EFLAGS. В системе команд сопроцессора существует команда **FSTSW**, которая позволяет запомнить значение регистра состояния сопроцессора в регистре AX или ячейке памяти. Далее значения нужных битов извлекаются и анализируются командами основного процессора. Обычным приемом здесь является запись старше-

го байта регистра состояния, в котором и находятся биты C0...C3, в младший байт регистра EFLAGS/FLAGS, осуществляемая командой **SAHF**. Эта команда записывает содержимое AH в младший байт регистра EFLAGS. В результате этого бит C0 записывается на место флага CF, C2 – на место PF, C3 – на место ZF (см. таблицу).

Пример. Поиск большего из двух чисел, находящихся в ST(0) и ST(1). Результат возвращается в ST(0).

```
FCOM            ; сравниваются ST(0) и ST(1)
FSTSW AX        ; после сравнения флаги сохраняются в регистре AX
SAHF            ; флаги загружаются в регистр флагов EFLAGS
JNC EXIT        ; если ST(0)>ST(L), т.е. CF=0, то на вершине стека
                ; находится большее число
FXCH ST(1)      ; ST(0) и ST(1) меняются операндами и на вершине стека
                ; оказывается большее число
EXIT:
. . . . . .
```

В новых микропроцессорах появилась группа команд сравнения **FCOMI**, **FCOMIP**, которые выполняют те же операции, что и FCOM, FCOMP, но в отличие от последних не требуют специальных команд для обработки битов C0…C3 регистра SWR, а устанавливают флаги ZF, CF и PF непосредственно в регистре флагов EFLAGS центрального процессора. Формат команд (операнды обязательны):

```
FCOMI ST, ST(i)
```

Пример. В предыдущем примере заменим команду FCOM командой

```
FCOMI ST, ST(1)
```

В результате последующие две команды (FSTSW и SAHF) становятся ненужными.

Арифметические команды. Команды сопроцессора, входящие в группу вещественных арифметических команд, реализуют четыре основные арифметические операции: сложение, вычитание, умножение и деление.

Общий формат команд:

```
Код_команды Приемник, Источник
```

Традиционные команды действуют по формуле

Приемник = Приемник *операция* Источник

Реверсивные команды для вычитания и деления действуют по формуле

Приемник = Источник *операция* Приемник

Форматы традиционных команд:

```
Fx ST, ST(i)      ; ST(0) = ST(0) операция ST(i)
Fx ST(i), ST      ; ST(i) = ST(i) операция ST(0)
FxP ST(i), ST     ; ST(i) = ST(i) операция ST(0), после удаления источника
                  ; из стека результат оказывается в ST(i-1)
Fx Mem            ; ST(0) = ST(0) операция Mem
FxP               ; ST(1) = ST(1) операция ST(0), после удаления источника
                  ; из стека результат оказывается в ST(0)
```

Форматы реверсивных команд:

```
FxR ST, ST(i)     ; ST(0) = ST(i) операция ST(0)
```

FxR ST(i), ST ; ST(i) = ST(0) операция ST(i)
FxRP ST(i), ST ; ST(i) = ST(0) операция ST(i), после удаления
; источника из стека результат оказывается в ST(i-1)
FxR Mem ; ST(0) = Mem операция ST(0)
FxRP ; ST(1) = ST(0) операция ST(1), после удаления источника
; из стека результат оказывается в ST(0)

Здесь x – модификатор команды: **ADD** для сложения, **SUB** для вычитания, **MUL** для умножения, **DIV** для деления, Mem – источник в памяти. Для команд вычитания и деления допустима буква R определяющая реверсивное действие команды. Буква P в конце названия определяет завершающее действие команды в виде удаления операнда-источника из стека.

Пример. Команда FADDP без операндов прибавляет содержимое ST(0) к ST(1), помещает результат в ST(1) и удаляет содержимое ST(0) из стека, так что результат оказывается в ST(0).

```
.DATA
OP1 DD 20.0
OP2 DD 100.0
.CODE
FLD OP1 ;  ST(0)=20.0
FLD OP2 ;  ST(0)=100.0, ST(1)=20.0
FADDP ;    ST(0)=120.0
```

Дополнительные арифметические команды (не имеют операндов, аргумент берется из регистра ST(0), туда же помещается результат):

FSQRT – команда вычисления квадратного корня;
FABS – команда вычисления модуля;
FCHS – команда изменения знака.

Команды тригонометрических функций. Команды не имеют операндов.

FCOS – команда вычисляет косинус угла, находящегося в вершине стека сопроцессора. Результат возвращается в регистр ST(0).
FSIN – команда вычисляет синус угла, находящегося в вершине стека сопроцессора. Результат возвращается в регистр ST(0).
FSINCOS – команда вычисляет синус и косинус угла, находящегося в вершине стека сопроцессора. Синус помещается в ST(1), а косинус – в ST(0).
FPTAN – команда вычисляет тангенс угла, находящегося в вершине стека сопроцессора. Результат возвращается в регистре ST(1), а в регистр ST(0) помещается единица.

Аргументы всех этих команд должны задаваться в радианах и не могут быть больше 2^{63} и меньше -2^{63}.

FPATAN – команда вычисляет арктангенс числа, получаемого при делении ST(1) на ST(0), сохраняет результат в ST(1) и выталкивает ST(0) из стека. Результат всегда имеет тот же знак, что и ST(1), и меньше π по абсолютной величине.

Команды вычисления логарифмов и степеней.

F2XM1 – команда вычисления значения функции $y = 2^x - 1$. Исходное значение x размещается в регистре ST(0) и должно лежать в пределах $\bullet 1 \bullet x \bullet 1$. Результат замещает x в регистре ST(0).

FYL2X – команда вычисления значения функции $z = y \cdot \log_2(x)$. Исходное значение x размещается в регистре ST(0) и должно быть неотрицательным ($0 \bullet x \bullet \bullet\bullet$), исходное значение y размещается в регистре ST(1). Перед тем как осуществить запись результата z в регистр ST(0), команда FYL2X выталкивает значения x и y из стека.

Используя тождества $\log_q(x) = \log_2(x) \cdot \log_q(2) = \log_2(x)/\log_2(q)$, можно вычислить логарифм любого числа по любому основанию. Значения констант lg(2), ln(2), $\log_2(10)$, $\log_2(e)$ загружаются соответствующими командами сопроцессора: FLDLG2, FLDLN2, FLDL2T, FLDL2E.

FYL2XP1 – команда вычисления значения функции $z = y \cdot \log_2(x + 1)$. Исходное значение x размещается в регистре ST(0), а исходное значение y – в регистре ST(1). Значение x должно лежать в диапазоне $0 \bullet |x| \bullet 1 \bullet \sqrt{2}/2$. Перед тем как записать результат z в регистр ST(0), команда FYL2XP1 выталкивает из стека значения x и y.

Пример. Вычисление натурального логарифма ln(x).

```
;              ST(0)          ST(1)
FLD1 ;         1
FLD X ;        X              1
FYL2X;         log2(x)
FLDL2E ;       log2(e)        log2(x)
FDIVP ;        ln(x)
```

Интерфейс с языком C++. Основной вопрос взаимодействия языков ассемблера и C++ касается передачи и возвращения аргументов.

Передача аргументов в подпрограмму происходит через стек программы (не сопроцессора). Числа одинарной (float) и двойной (double) точности помещаются в стек без каких-либо преобразований. Числа расширенной точности в Visual C++ не используются, тип long double здесь совпадает по формату с double.

При возврате из подпрограммы на ассемблере в программу на C++ необходимо очистить стек сопроцессора, но если результат возвращается через имя подпрограммы (как значение типа float или double), то необходимо поместить его в ST(0).

Кроме того, подпрограмма на ассемблере должна сохранять значение регистра управления CWR программы на C++.

4. Варианты заданий

Вычислить значение функции *f(x)* в произвольной точке x.

№	*f(x)*
1	$2\sin^2(x) / 3$
2	$x + \ln(x+0.5)$

3	$tg(x) - x$
4	$ctg(x) - x/3$
5	$(x + 2x^2) / (x - 1)$
6	$3\cos^2(x) / 4$
7	$x*\ln(x)$
8	$tg(x)/2$
9	$ctg(x) - 2x$
10	$x^3 / (x^2 + 1)$
11	$1 / \sin(x)$
12	$2 / \cos(x)$

5. Контрольные вопросы

1. Форматы и способы задания целых и двоично-десятичных чисел.
2. Форматы и способы задания вещественных чисел.
3. Денормализованные вещественные числа, нуль, бесконечность.
4. Нечисла (SNAN, QNAN).
5. Регистровый стек: принципы работы.
6. Регистр состояния SWR: назначение, основные поля.
7. Регистр управления CWR: назначение, основные поля.
8. Регистр тегов TWR: назначение, принцип использования.
9. Особенности вычисления выражений с использованием постфиксной записи.
10. Инициализация сопроцессора: команда FINIT.
11. Команды передачи данных в вещественном формате: FLD, FST(P), команда обмена FXCH.
12. Команды передачи данных в целочисленном и двоично-десятичном формате: FILD, FIST(P), FBLD, FBSTP.
13. Команды сравнения данных: FCOM, FCOMP(P), FICOM(P), FTST, FCOMI(P).
14. Целочисленные арифметические команды: FIADD, FISUB(R), FIMUL, FIDIV(R).
15. Вещественные арифметические команды: Fx(P), FxR(P).
16. Дополнительные арифметические команды: FSQRT, FABS, FCHS, FXTRACT, FSCALE, FRNDINT.
17. Дополнительные арифметические команды: FPREM, FPREM1
18. Команды тригонометрических функций: FCOS, FSIN, FSINCOS, FPTAN, FPATAN.
19. Команды вычисления логарифмов и степеней: F2XM1, FYL2X, FYL2XP1.
20. Команды управления сопроцессором: FINCSTP, FDECSTP, FFREE.
21. Исключения сопроцессора и их обработка: недействительная операция.
22. Исключения сопроцессора и их обработка: деление на ноль, денормализация операнда.
23. Исключения сопроцессора и их обработка: переполнение и антипереполнение, неточный результат.
24. Исключения сопроцессора и их обработка: приоритет особых случаев.

Литература

1. Абель П. Язык Ассемблера для IBM PC и программирование. – М.: Высш. шк., 1992.

2. Вержбицкий В.М. Основы численных методов. – М.: Высш. шк., 2002.

3. Вержбицкий В.М. Численные методы (линейная алгебра и нелинейные уравнения). – М.: Высш. шк., 2000.

4. Григорьев В.Л. Архитектура и программирование арифметического сопроцессора. – М.: Энергоатомиздат, 1991.

5. Григорьев В.Л. Микропроцессор i486. Архитектура и программирование (в 4-х книгах). – М.: ГРАНАЛ, 1993.

6. Зубков С.В. Assembler для DOS, Windows и Unix. – М.: ДМК Пресс, 2000.

7. Ирвин К. Язык ассемблера для процессоров Intel. – М.: Издательский дом «Вильямс», 2005.

8. Магда Ю.С. Ассемблер для процессоров Intel Pentium. – СПб.: Питер, 2006.

9. Мэтьюз Дж.Г., Финк К.Д. Численные методы. Использование MATLAB. – М.: Издательский дом «Вильямс», 2001.

10. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: ДИАЛОГ-МИФИ, 1999.

11. Пирогов В.Ю. Ассемблер на примерах. – СПб.: БХВ-Петербург, 2005.

12. Пирогов В.Ю. Ассемблер для Windows. – СПб.: БХВ-Петербург, 2007.

13. Программирование на языке Ассемблера: методические указания к лабораторным работам по курсам "Языки программирования и методы трансляции" (для 3 курса ФПМИ по направлению 010400 – Прикладная математика и информатика) и "Теория вычислительных процессов и структур" (для 3 курса ФПМИ по направлению 010500 – Математическое обеспечение и администрирование информационных систем) / Новосиб. гос. техн. ун-т; сост.: Д. В. Лисицин, Р. В. Петров, И. А. Полетаева. – Новосибирск, 2011..

14. Секунов Н. Ю. Самоучитель Visual C++.NET. – СПб.: БХВ-Петербург, 2002.

15. Скенлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке Ассемблера. – М.: Радио и связь, 1989.

16. Трой Д. Программирование на языке Си для персонального компьютера IBM PC. – М.: Радио и связь, 1991.

17. Юров В.И. Assembler. Учебник для вузов. – СПб.: Питер, 2007.

18. Юров В.И. Assembler. Практикум. – СПб.: Питер, 2007.