Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра информационных технологий автоматизированных систем

В.Н. Лепешинский

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по курсу

МИКРОПРОЦЕССОРЫ И МИКРОКОМПЬЮТЕРЫ

для студентов специальности 53 01 02
«Автоматизированные системы обработки информации»
дневной формы обучения

Минск 2002

УДК 681.325.5-181.48 (075.8)
ББК 32.973.26-04 я 73
Л 48

Лепешинский В.Н.

Л 48 Лабораторный практикум по курсу «Микропроцессоры и микрокомпьютеры» для студентов специальности 53 01 02 «Автоматизированные системы обработки информации» дневной формы обучения / В.Н. Лепешинский. — Мн.: БГУИР, 2002. — 68 с.: ил.

ISBN 985-444-363-9

Лабораторный практикум составлен на основе конспекта лекций по курсу «Микропроцессоры и микрокомпьютеры» для студентов специальности 53 01 02 «Автоматизированные системы обработки информации» дневной формы обучения и содержит шесть лабораторных работ по вопросам программирования на языке ассемблер.

УДК 681.325.5-181.48 (075.8)
ББК 32.973.26-04 я 73

ISBN 985-444-363-9

СОДЕРЖАНИЕ

Лабораторная работа № 1. Основы программирования на языке ассемблер
Лабораторная работа № 2. Основы программирования для MS-DOS
Лабораторная работа № 3. Использование цепочечных команд
Лабораторная работа № 4. Совместное программирование на языках ассемблер и C/C++. Программирование на ассемблере в ОС WINDOWS
Лабораторная работа № 5. Программирование математического сопроцессора
Лабораторная работа № 6. Программирование с использованием технологии MMX
Литература

Лабораторная работа № 1

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ АССЕМБЛЕР

Цели работы: ознакомиться с основами программирования на языке ассемблер. Научиться создавать, компилировать и компоновать программы на языке ассемблер с помощью пакета TASM.

1 Программная модель 32-разрядных микропроцессоров

1.1 Набор регистров

В программах на языке ассемблер регистры микропроцессора (МП), изображенные на рисунке 1.1, используются очень интенсивно.

Рисунок 1.1 — Основные регистры 32-разрядных процессоров

1.1.1 Регистры общего назначения (РОН) без каких-либо ограничений могут использоваться для хранения операндов логических и арифметических операций, компонентов адреса, указателей на ячейки памяти:

EAX/AX/AH/AL (*Accumulator register*) — *аккумулятор* — применяется для хранения промежуточных данных;

EBX/BX/BH/BL (*Base register*) — *базовый регистр* — применяется для хранения базового адреса некоторого объекта в памяти;

ECX/CX/CH/CL (*Count register*) — *регистр-счетчик* — применяется в командах, производящих некоторые повторяющиеся действия;

EDX/DX/DH/DL (*Data register*) — *регистр данных* — так же, как и регистр *EAX/AX/AH/AL*, он хранит промежуточные данные.

Следующие два регистра используются для поддержки цепочечных команд (подробно рассматриваются в лабораторной работе № 3):

ESI/SI (*Source Index register*) — *индекс источника* — содержит текущий адрес элемента в цепочке-источнике;

EDI/DI (*Destination Index register*) — *индекс приемника* — содержит текущий адрес элемента в цепочке-приемнике.

Следующие регистры предназначены для работы со стеком:

ESP/SP (*Stack Pointer register*) — *регистр указателя стека* — содержит указатель вершины стека в сегменте стека. Этот регистр не следует использовать явно для хранения каких-либо операндов программы;

EBP/BP (*Base Pointer register*) — *регистр указателя базы кадра стека* — предназначен для организации произвольного доступа к данным внутри стека.

1.1.2 Сегментные регистры предназначены для аппаратной поддержки структурной организации программы в виде отдельных частей, называемых *сегментами*. Сегмент представляет собой независимый блок памяти фиксированного размера. Операционная система размещает сегменты программы в оперативной памяти, после чего помещает их адреса в соответствующие сегментные регистры.

МП поддерживает следующие типы сегментов:

сегмент кода — содержит машинные команды, для доступа к нему служит регистр *CS* (*Code Segment register*) — *сегментный регистр кода*;

сегмент стека — для доступа к нему служит регистр *SS* (*Stack Segment register*) — *сегментный регистр стека*;

сегмент данных — содержит обрабатываемые программой данные, для доступа к нему служит регистр *DS* (*Data Segment register*) — *сегментный регистр данных*;

дополнительные сегменты данных — их адреса должны содержаться в регистрах *ES*, *GS*, *FS*.

1.1.3 Регистры состояния и управления содержат информацию о состоянии МП и программы. К этим регистрам относятся: *регистр флагов EFLAGS/FLAGS* и *регистр указатель команд EIP/IP*. Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние МП. На рисунке 1.2 показано содержимое регистра *EFLAGS*, а в таблице 1.1 — назначение отдельных флагов.

← EFLAGS →																							
								← FLAGS →															
31	…	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
0	…	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Рисунок 1.2 — Содержимое регистра EFLAGS

Указатель команд *EIP/IP* содержит адрес следующей выполняемой команды относительно содержимого регистра *CS*. Регистр *EIP/IP* непосредственно недоступен программисту, но загрузка и изменение его значения производятся командами условных и безусловных переходов, вызова и возврата из процедур, а также вызова и возврата из прерываний.

Таблица 1.1 — Назначение флагов регистра EFLAGS

Флаг	Название	Назначение
1	2	3
CF	Флаг переноса	= 1 — арифметическая операция произвела перенос из старшего бита результата = 0 — переноса не было
PF	Флаг паритета	= 1 — 8 младших разрядов результата содержат четное число единиц = 0 — нечетное
AF	Дополнительный флаг переноса	Для команд, работающих с BCD-числами: = 1 — перенос из разряда 3 в 4-й при сложении или заем в разряд 3 из 4-го при вычитании = 0 — не было переносов или заемов
ZF	Флаг нуля	= 1 — результат нулевой = 0 — результат ненулевой
SF	Флаг знака	= 1 — старший бит результата равен 1 = 0 — старший бит результата равен 0
TF	Флаг трассировки	При пошаговой работе МП: = 1 — МП генерирует прерывание с номером 1 после выполнения каждой машинной команды = 0 — обычная работа
IF	Флаг прерывания	= 1 — аппаратные прерывания разрешены = 0 — аппаратные прерывания запрещены
DF	Флаг управления	Определяет направление обработки цепочек: = 1 — от конца к началу = 0 — от начала к концу
OF	Флаг переполнения	= 1 — произошел перенос в старший или заем из старшего знакового бита результата = 0 — не было переноса или заема
IOPL	Уровень привилегий ввода-вывода	Для контроля доступа к командам ввода-вывода в защищенном режиме работы МП
NT	Флаг вложенности задачи	Для фиксации в защищенном режиме работы МП того факта, что одна задача вложена в другую
RF	Флаг возобновления	Используется при обработке прерываний от регистров отладки
VM	Флаг виртуального 8086	= 1 — МП в режиме виртуального 8086 = 0 — МП в реальном или защищенном режиме
AC	Флаг контроля выравнивания	Разрешение контроля выравнивания при обращениях к памяти
VIF	Флаг виртуального прерывания	Появился в МП Pentium. В V-режиме является аналогом флага IF

Окончание таблицы 1.1

1	2	3
VIP	Флаг отложенного виртуального прерывания	Появился в МП Pentium. = 1 — отложенное прерывание, используется при работе в V-режиме совместно с флагом VIF
ID	Флаг идентификации	Если программа может установить этот флаг, то МП поддерживает инструкцию CPUID

1.2 Типы данных

С точки зрения размерности, МП аппаратно поддерживает следующие основные типы данных, изображенные на рисунке 1.3:

байт — восемь последовательно расположенных бит;

слово — два байта, имеющих последовательные адреса. Слово делится на *младший байт* и *старший*. Младший байт всегда хранится по меньшему адресу, который является *адресом слова*;

двойное слово — четыре байта, расположенных по последовательным адресам. Двойное слово состоит из *младшего слова* и *старшего*. Младшее слово хранится по меньшему адресу, который является *адресом двойного слова*;

учетверенное слово — восемь байт, расположенных по последовательным адресам. Учетверенное слово делится на *младшее двойное слово* и *старшее двойное слово*. Младшее двойное слово хранится по меньшему адресу, который является *адресом учетверенного слова*;

128-битный упакованный тип данных — появился в МП Pentium III. Для работы с ним были введены специальные команды.

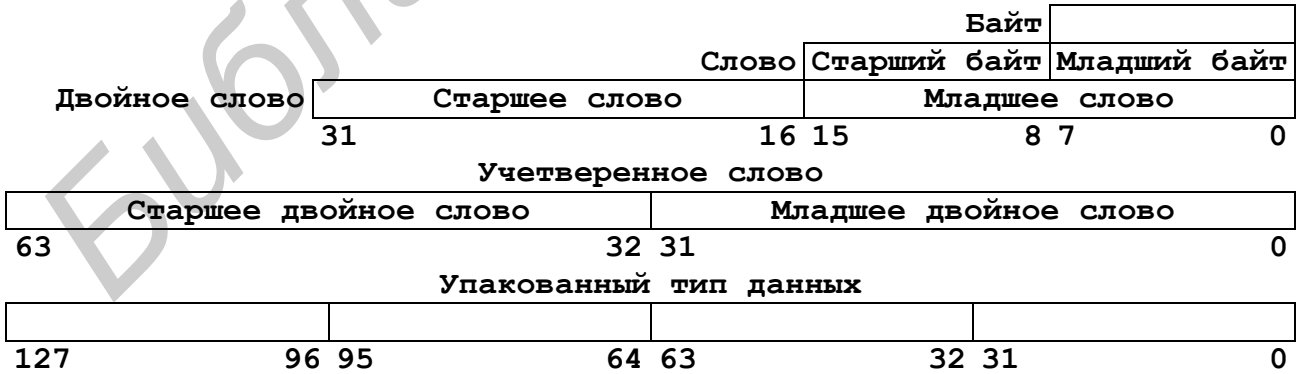

Рисунок 1.3 — Основные типы данных микропроцессора

Кроме трактовки типов данных с точки зрения их разрядности, МП поддерживает их логическую интерпретацию, изображенную на рисунке 1.4:

целый тип со знаком — двоичное значение со знаком размером 8/16/32 бита. Знак содержится соответственно в 7/15/31 бите (ноль — положительное число, единица — отрицательное). Отрицательные числа представляются в дополнительном коде. Числовые диапазоны приведены в таблице 1.2;

целый тип без знака — двоичное значение со знаком размером 8/16/32 бита. Числовые диапазоны для этого типа данных приведены в таблице 1.2;

ближний указатель — 32-разрядный логический адрес, представляющий собой относительное смещение в байтах от начала сегмента;

дальний указатель — 48-разрядный логический адрес, состоящий из двух частей: 16-разрядной сегментной части — *селектора* и 32-разрядного смещения;

цепочка — некоторый непрерывный набор байт, слов или двойных слов длиной до 4 Гбайт (подробно рассматривается в лабораторной работе № 3);

битовое поле — непрерывная последовательность бит. Каждый бит является независимым и может рассматриваться как отдельная переменная;

неупакованный двоично-десятичный тип — байтовое представление десятичной цифры от 0 до 9. Числа хранятся как байтовые значения без знака по одной цифре в каждом байте (в младшей тетраде);

упакованный двоично-десятичный тип — представление двух десятичных цифр от 0 до 9. Каждая цифра хранится в своей тетраде (старшая — в старшей, младшая — в младшей);

типы данных с плавающей точкой — специальные типы данных для обработки чисел с плавающей точкой в математическом сопроцессоре (подробно рассматриваются в лабораторной работе № 5);

типы данных MMX-расширения (Pentium MMX/II) — представляют собой совокупность упакованных целочисленных элементов определенного размера (подробно рассматриваются в лабораторной работе № 6);

типы данных XMM-расширения (Pentium III) — представляют собой совокупность упакованных элементов с плавающей точкой фиксированного размера.

Таблица 1.2 — Диапазоны целых чисел

Размерность, бит	Без знака	Со знаком
8	0 — 255	–128 — +127
16	0 — 65 535	–32 768 — +32 767
32	0 — 4 294 967 295	–2 147 483 648 — +2 147 483 647

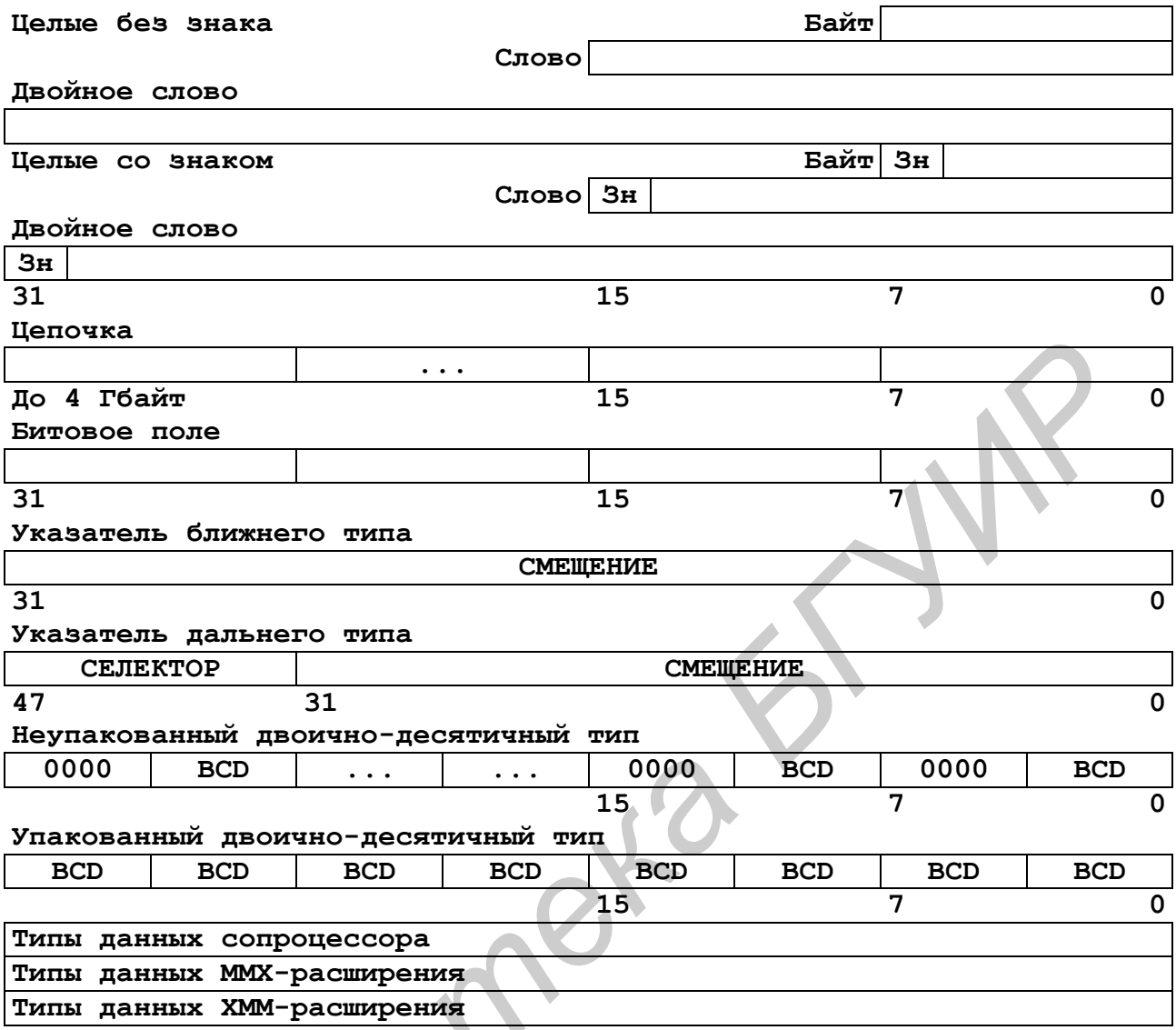

Рисунок 1.4 — Основные логические типы данных микропроцессора

2 Введение в язык ассемблер

2.1 Структура программы

Программа на языке ассемблера состоит из строк, содержащих следующие поля:

метка: команда/директива операнды ;комментарий

Все эти поля необязательны. Метка может быть любой комбинацией символов английского алфавита, цифр и символов «_», «$», «@», «?». Цифра не может быть первым символом метки. Большие и маленькие буквы по умолчанию не различаются, но различие можно включить, задав соответствующую опцию в командной строке ассемблера. В поле команды может располагаться команда процессора, которая будет транслирована в исполнимый код, или ди-

ректива, которая не приводит к генерации кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды. Текст от символа «**;**» не анализируется ассемблером и используется в качестве комментария.

Если метка располагается перед командой процессора, сразу после нее ставится символ «**:**», например:

```
some_loop:
      lodsw          ;считать слово из строки в ax
      cmp    ax, 7   ;если это 7 - выйти из цикла
      loopne some_loop
```

Если метка стоит перед директивой — двоеточие не ставится. Рассмотрим директивы, работающие напрямую с метками и их значениями.

Директива *label* определяет идентификатор *name* и задает его тип *type*:

```
name       label  type
```

Тип может быть одним из следующих: *byte* — байт, *word* — слово, *dword* — двойное слово, *fword* — 6 байт, *qword* — учетверенное слово, *tbyte* — 10 байт, *near* — ближняя метка, *far* — дальняя метка. Идентификатор получает значение, равное адресу следующей команды или следующих данных. С помощью директивы *label* удобно организовывать доступ к одним и тем же данным, как к байтам, так и к словам, определив перед данными две метки с разными типами.

Директива *equ* присваивает идентификатору *name* значение, которое определяется как результат выражения *expression*:

```
name       equ    expression
```

Это может быть целое число, адрес или любая строка символов, например:

```
truth     equ  1
message1  equ    'Try again$'
var2      equ    [si + 4]
          cmp    ax, truth ;cmp  ax, 1
          db     message1      ;db   'Try again$'
          mov    ax, var2      ;mov  ax, [si + 4]
```

Директива *equ* чаще всего используется для введения параметров, общих для всей программы.

С помощью директивы = идентификатор может принимать только целочисленные значения. Кроме того, идентификатор может быть переопределен позже, например:

```
buf_size = 1024
```

2.2 Директивы распределения памяти

2.2.1 Определение переменных в общем виде записывается следующим образом:

```
name        d*      value
```

где *d** — одна из псевдокоманд: *db* — определить байт, *dw* — определить слово (2 байта), *dd* — определить двойное слово (4 байта), *df* — определить 6 байт (дальний указатель), *dq* — определить учетверенное слово (8 байт), *dt* — определить 10 байт (80-битные типы данных, используемые FPU).

Поле значения может содержать одно или несколько чисел, символов строк (взятых в одиночные или двойные кавычки), операторов *?* и операторов *dup*, разделенных запятыми, например:

```
text_string      db     'Hello world!', 0
number           dw     7
table            db     0,1,2,3,4,5,6,7,8,9
float_number     dd     3.5e7
```

Если вместо точного значения указан знак *?*, переменная считается неинициализированной, например:

```
var1             dq     ?
```

Если требуется заполнить участок памяти повторяющимися данными, используется специальный оператор *dup*, имеющий формат

```
count dup (value)
```

Например, определение

```
table_512wdw     512 dup(?)
```

создает массив из 512 неинициализированных слов.

2.2.2 Структуры объявляются с помощью директивы *struc*, которая позволяет определить структуру данных аналогично структурам в языках высокого уровня:

```
struct_name   struc
    fields
ends
```

где *fields* — любой набор псевдокоманд определения переменных или структур. В дальнейшем для создания объектов такой структуры в памяти используют имя структуры:

```
name      struct_name   <values>
```

Для чтения и записи в элемент структуры используется оператор «.», например:

```
point struc                ;определение структуры
   x      dw  0          ;два слова со значениями
   y      dw  0          ;по умолчанию 0, 0
   color  db  3 dup(?)      ;и три байта для цвета
ends
cur_point point  <1,1,1,255,255,255> ;инициализация
...
   mov ax, cur_point.x  ;обращение к слову x
```

2.3 Организация программы на ассемблере

2.3.1 Набор допустимых команд указывается в самом начале программы. По умолчанию ассемблер использует набор команд процессора 8086 и выдает сообщения об ошибках, если встречается команда, которую этот процессор не поддерживает. Для разрешения использования команд более новых процессоров предлагаются следующие директивы:

```
.186          ;команды 80186;
.286          ;непривилегированные команды 80286;
.386          ;непривилегированные команды 80386;
.486          ;непривилегированные команды 80486;
.586          ;непривилегированные команды P5 (Pentium);
.686          ;непривилегированные команды P6 (Pentium);
.8087      ;команды FPU 8087;
.287          ;команды FPU 80287;
.387          ;команды FPU 80387;
.487          ;команды FPU 80486;
.587          ;команды FPU 80586;
.MMX          ;команды IA MMX;
.K3D          ;команды AMD 3D;
```

2.3.2 Модель памяти в общем виде задается директивой

```
.model model_name, language, modifier
```

где *model_name* — одно из значений, перечисленных в таблице 2.1; *language* — необязательный операнд, принимающий значения *C*, *PASCAL*, *BASIC*, *FORTRAN*, *SYSCALL* и *STDCALL*. Если он указан, ассемблер считает, что все процедуры рассчитаны на вызов из программ на соответствующем языке высо-

кого уровня. *Modifier* — необязательный операнд, принимающий значения *NEARSTACK* (по умолчанию) или *FARSTACK*.

Таблица 2.1 — Модели памяти

Модель	Описание
TINY	Код, данные и стек размещаются в одном и том же сегменте размером 64Кбайт
SMALL	Код — в одном сегменте, а данные и стек — в другом
COMPACT	Код — в одном сегменте, а данные могут располагаться в нескольких сегментах. Используются дальние указатели
MEDIUM	Код — в нескольких сегментах, а все данные — в одном. Для доступа к данным используется только смещение, а для вызова подпрограмм — команды дальнего вызова процедур
LARGE, HUGE	И код, и данные могут занимать несколько сегментов
FLAT	То же, что и *TINY*, но используются 32-разрядные адреса. Максимальный размер сегмента — 4 Мбайт

2.3.3 Директивы сегментации определяют сегменты программы. Как отмечалось ранее, каждая программа состоит из сегмента кода, сегмента данных и сегмента стека.

Сегменты описываются директивами *segment* и *ends*:

```
segment_name     segment    attributes
    ...
ends
```

здесь *attributes* — атрибуты сегмента. На практике чаще всего используют упрощенные директивы определения сегментов:

```
.stack size      ;сегмент стека в size байт
                 ;(по умолчанию 1024)
.data            ;обычный сегмент данных
.const           ;сегмент неизменяемых данных
.code            ;основной сегмент кода
```

2.3.4 Глобальные объявления необходимы для определения идентификаторов, общих для разных модулей. Директива

```
public language      name
```

делает идентификатор *name* доступным из других модулей программы. Директива

```
extrn  language      name:type
```

описывает идентификатор *name*, определенный в другом модуле (с помощью *public*). Тип *type* должен соответствовать типу идентификатора в том модуле, в котором он был определен. Директива

```
global language      name:type
```

действует, как *public* и *extrn* одновременно. Если идентификатор *name* находится в том же модуле, он становится доступным для других модулей (действует как *public*). Если идентификатор не описан, то он считается внешним и выполняется действие, аналогичное действию директивы *extrn*.

2.4 Выражения

Выражение — это набор идентификаторов, чисел или строк, связанных друг с другом операторами. Все выражения вычисляются в ходе ассемблирования программы, так что в полученном коде используются только значения.

В выражениях используются операторы следующих типов:

арифметические операторы: + (плюс), – (минус), * (умножение), / (целочисленное деление), *mod* (остаток от деления), например:

```
mov al, 90 mod 7     ; mov  al, 6
```

логические операторы: *and* (И), *not* (НЕ), *or* (ИЛИ), *xor* (исключающее ИЛИ), *shl* (сдвиг влево), *shr* (сдвиг вправо), например:

```
mov ax, 1234h and 4321h ; mov  ax, 0220h
```

операторы сравнения: *eq* (равно), *ge* (больше или равно), *gt* (больше), *le* (меньше или равно), *lt* (меньше), *ne* (не равно). Результат действия каждого из этих операторов — единица, если условие выполняется, и ноль — если не выполняется;

операторы адресации: *seg expr* — сегментный адрес *expr*; *offset expr* — смещение *expr*; *type ptr expr* — переопределение типа *expr* в соответствии с типом *type*; *sizeof var/type* — число байт, занимаемых переменной *var* или типом *type*, например:

```
mov dx, offset msg      ;занести в dx
                      ;смещение переменной msg
mov ax, sizeof word     ;занести в ax размер типа word
mov dword ptr [si], 0      ;записать 4 байта нулей
                      ;по адресу ds:si
```

другие операторы: *()* (круглые скобки) — задают приоритет вычислений, *[]* (квадратные скобки) — косвенная адресация.

2.5 Другие директивы

Директива

```
include       file_name
```

вставляет в текст программы текст файла аналогично команде препроцессора языка С *#include*. Обычно используется для включения файлов, содержащих определения констант, структур и макросов. Директива

```
includelib    file_name
```

указывает компоновщику имя дополнительной библиотеки или объектного файла, который потребуется для компоновки данной программы.

2.6 Формат команд и режимы адресации

В машинную команду МП явно или неявно входят следующие элементы:

поле префиксов — элемент команды, который уточняет либо модифицирует действие этой команды в следующих аспектах: замена сегмента, изменение размерности адреса, изменение размерности операнда, указание на необходимость повторения данной команды;

поле кода операции, определяющее действие данной команды;

поле операндов; содержит от 0 до 3 элементов.

Важной особенностью машинных команд является то, что они не могут манипулировать одновременно двумя операндами, находящимися в оперативной памяти. По этой причине возможны только следующие сочетания операндов в команде:

регистр — регистр;
регистр — память;
память — регистр;
регистр — непосредственный операнд;
память — непосредственный операнд.

Из перечисленных сочетаний операндов наиболее часто употребляются *регистр — память* и *память — регистр*. Ввиду их важности рассмотрим их подробнее, и в частности способы адресации операндов в памяти.

2.6.1 Прямая адресация — простейший вид адресации операнда в памяти. Прямая адресация может быть двух типов:

относительная — используется в командах условного и безусловного перехода, например:

```
    jz  mm1     ;переход на метку mm1, если флаг zf = 1
    mov al, 32
...
```

```
mm1:
```

абсолютная — использует несколько форм такой адресации, например:

```
    mov ax, dword ptr [0000] ;записать слово по адресу
                             ;ds:0000 в регистр ax
```

Но такая адресация применяется редко, обычно используемым ячейкам в программе присваиваются символические имена, например:

```
.data
var1       dw  20h
...
    add var1, ax   ;сложить содержимое переменной var1 и
                   ;регистра ax, результат поместить в
                   ;переменную var1 (ее адрес ds:0000)
```

2.6.2 Косвенная базовая (регистровая) адресация — адрес операнда находится в любом из регистров общего назначения, кроме *SP/ESP* и *BP/EBP*. Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки *[]*. Например, команда

```
mov bx, [eax]
```

помещает в регистр *BX* содержимое слова по адресу из сегмента данных со смещением, хранящимся в регистре *EAX*.

Этот режим очень удобен для организации циклических вычислений и для работы с различными структурами данных типа таблиц и массивов.

2.6.3 Косвенная базовая (регистровая) адресация со смещением является дополнением предыдущей и предназначена для доступа к данным с известным смещением относительно некоторого базового адреса. Например, команда

```
mov bx, [eax + 10h]
```

пересылает в регистр *BX* слово из области памяти по адресу: содержимое *EAX + 10h*. Следующий пример иллюстрирует инициализацию массива последовательными значениями:

```
mas    db  5 dup(0)      ;массив из 5 слов
...
    mov edi, 5
mm1:
    dec eax           ;eax = eax - 1
    mov mas[eax], al      ;mas[eax] = al
    jnz mm1    ;повторяем, пока eax не станет равным 0
```

2.6.4 Косвенная индексная адресация со смещением очень похожа на косвенную базовую адресацию со смещением, но обладает той особенностью, что позволяет *масштабировать* содержимое индексного регистра. Например, в команде

```
mov ax, mas[esi*2]
```

значение эффективного адреса второго операнда вычисляется выражением *MAS + (ESI) * 2*. Возможность масштабирования существенно помогает для организации индексации массивов элементов размером 1, 2, 4 или 8 байт.

2.6.5 Косвенная базовая индексная адресация похожа на косвенную базовую адресацию со смещением. Эффективный адрес формируется как сумма содержимого двух любых регистров общего назначения: базового и индексного. Например:

```
mov eax, [esi][edx] ;эффективный адрес = (esi)+(edx)
```

2.6.6 Косвенная базовая индексная адресация со смещением является дополнением косвенной базовой индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения смещения. Например, команда

```
mov eax, [esi + 5][edx]
```

пересылает в регистр *eax* двойное слово по адресу: *(ESI) + 5 + (EDX)*, а команда

```
add ax, array[esi][ebx]
```

производит сложение содержимого регистра *AX* с содержимым слова по адресу: *ARRAY + (ESI) + (EBX)*.

3 Создание первой программы

Традиционно первая программа для освоения нового языка программирования — программа, выводящая на экран текст «*Hello world!*».

Наберите в любом текстовом редакторе следующий текст:

```
.model small
.stack
.data
Message   db 'Hello world!', 13, 10, '$'
.code
begin:
   mov  ax, @data ;@data – адрес сегмента данных
   mov  ds, ax    ;ds указывает на сегмент данных
```

```
    mov  ah, 9      ;функция печати строки DOS
    mov  dx, offset Message ;dx указывает на Message
    int  21h        ;вывод на дисплей «Hello world!»
    mov  ah, 4ch    ;функция завершения программы в DOS
    int  21h        ;завершить программу
end begin
```

Сохраните файл под именем *hello_1.asm*.

3.1 Ассемблирование программы

Прежде чем получить исполнимый файл, сначала нужно вызвать ассемблер, для того чтобы скомпилировать программу в объектный файл с именем *hello_1.obj*, набрав в командной строке следующую команду:

```
tasm hello_1.asm
```

С ассемблерными программами также можно работать из интегрированных сред разработки (Turbo C, Borland C++ 3.1), но создание полноценных программ на ассемблере требует некоторой их перенастройки.

3.2 Компоновка программы

После успешного ассемблирования следующим шагом является компоновка. Для компоновки используется компоновщик *tlink*. В командной строке необходимо ввести

```
tlink hello_1.obj
```

В результате при отсутствии ошибок будет получен исполнимый файл с именем *hello_1.exe*. Если его выполнить, на экране появится строка «*Hello world!*» и программа завершится.

3.3 Модификация первой программы

Изменим программу таким образом, чтобы она могла принять из внешнего мира байт входной информации (в данном случае с клавиатуры):

```
.model small
.stack
.data
TimePrmpt          db   'Уже день? (д/н)$'
```

```
GoodMorningMsg    db  'Доброе утро!', 13, 10, '$'
GoodAfternoonMsg  db  'Добрый день!', 13, 10, '$'
.code
start:
   mov  ax, @data   ;в ax адрес сегмента данных
   mov  ds, ax      ;ds указывает на сегмент данных
   mov  dx, offset TimePrmpt ;dx указывает на подсказку
   mov  ah, 9    ;функция вывода строки DOS
   int  21h         ;вывод на дисплей подсказки
   mov  ah, 1    ;функция приема символа DOS
   int  21h         ;прием одного символа
   cmp  al, 'д'     ;введен символ 'д'?
   jz   is_afternoon   ;да, время за полдень
   cmp  al, 'Д'     ;введен символ 'Д'
   jnz  is_morning  ;нет, время до полудня
is_afternoon:
   mov  dx, offset GoodAfternoonMsg ;дневное приветствие
   jmp  display_greeting
is_morning:
   mov  dx, offset GoodMorningMsg ; утреннее приветствие
display_greeting:
   mov  ah, 9;функция вывода строки DOS
   int  21h      ;вывод на дисплей приветствия
   mov  ah, 4ch  ;функция завершения программы в DOS
   int  21h      ;конец программы, выход в DOS
end start
```

4 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить программную модель 32-разрядного микропроцессора, набор регистров, типы данных;

б) изучить принципы построения программ на языке ассемблер, директивы, определение переменных, режимы адресации;

в) набрать в любом текстовом редакторе пример программы, приведенной в разд. 3. Сохранить в файле с расширением *.asm*;

г) полученный файл откомпилировать с помощью утилиты *tasm.exe*, а с помощью компоновщика *tlink.exe* получить исполнимый файл, как указано в подразд. 3.1 и 3.2;

д) то же самое проделать с примером программы из подразд. 3.3;

е) получить индивидуальное задание у преподавателя.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

Лабораторная работа № 2

ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ MS-DOS

Цели работы: изучить принципы программирования в ОС MS-DOS, научиться создавать EXE- и COM-программы, ознакомиться с основными функциями базовой системы ввода/вывода.

1 Программы типа COM

Файлы типа COM содержат только скомпилированный код без какой-либо дополнительной информации о программе. Весь код, данные и стек такой программы располагаются в одном сегменте и не могут превышать 64 Кбайт.

Рассмотрим типичный пример COM-программы:

```
;hello_1.asm
.model tiny      ;модель памяти, используется для COM
.code          ;начало сегмента кода
   org   100h;начальное значение счетчика
start:
   mov ah, 9  ;номер функции DOS – в ah
   mov dx, offset message  ;адрес строки – в dx
   int 21h    ;вызов системной функции DOS
   int 20h    ;завершение COM-программы
message   db 'Hello World!', 13, 10, '$';строка для
                                  ;вывода
   end start  ;конец программы
```

Чтобы получить исполнимый файл, сначала вызываем ассемблер:

```
tasm hello_1.asm
```

затем компоновщик:

```
tlink /t /x hello_1.obj
```

Рассмотрим исходный текст программы.

Модель памяти *tiny*, в которой сегменты кода, данных и стека объединены, предназначена для создания файлов типа COM.

org 100h устанавливает значение программного счетчика в 100h, так как при загрузке COM-файла в память DOS занимает первые 256 байт блоком данных PSP и располагает код программы только после этого блока. Все COM-программы должны начинаться с этой директивы.

Команда *mov ah, 9* помещает число 9 (номер функции DOS «вывод строки») в регистр *AH*.

Команда *mov dx, offset message* помещает в регистр *DX* смещение метки *MESSAGE* относительно начала сегмента данных, который в нашем случае совпадает с сегментом кода.

Команда *int 21h* вызывает системную функцию DOS с номером 9 (см. далее).

Команда *int 20h* используется для корректного завершения программы.

Следующая строка программы определяет строку данных «*Hello World*», управляющий символ ASCII «возврат каретки» (13), управляющий символ ASCII «перевод строки» (10) и символ «$», завершающий строку.

Последняя директива *end* завершает программу, одновременно указывая, с какой метки должно начинаться выполнение программы.

2 Программы типа EXE

EXE-программы немного сложнее в исполнении, чем COM-программы, но для них отсутствуют ограничения размера в 64 Кбайт. Рассмотрим пример EXE-программы:

```
;hello_2.asm
.model small
.stack
.data
   message   db 'Hello world!', 13, 10, '$'
.code
begin:
   mov ax, @data
   mov ds, ax     ;ds указывает на сегмент данных
   mov ah, 9      ;функция печати строки DOS
   mov dx, offset message ;dx указывает на Message
   int 21h        ;вывод на дисплей «Hello world!»
   mov ah, 4ch    ;функция завершения программы в DOS
   int 21h        ;завершить программу
end begin
```

В этом примере определяются три сегмента — сегмент стека, сегмент данных, содержащий строку, и сегмент кода.

При запуске EXE-программы регистр *DS* уже не содержит адреса сегмента со строкой *message* (он указывает на сегмент, содержащий блок данных PSP), а для вызова функции DOS этот регистр должен иметь сегментный адрес строки. Команда *mov ax, @data* загружает в *AX* адрес сегмента данных, а *mov ds, ax* копирует его в *DS*. И наконец, программы типа EXE должны завершаться вызовом системной функции DOS с номером 4ch.

Компиляция *hello_2.asm*:

```
tasm hello_2.asm
tlink /x hello_2.obj
```

3 Прерывания базовой системы ввода/вывода

3.1 Понятие прерывания

В современных процессорах принят подход, основанный на понятии *прерывания*. Прерывание — инициируемый определенным образом процесс, временно переключающий микропроцессор на выполнение другой программы с последующим возобновлением выполнения прерванной программы. Это позволяет обеспечить наиболее эффективное управление не только внешними устройствами, но и программами. Например, нажатие клавиши на клавиатуре инициирует немедленный вызов программы, которая распознает нажатую клавишу, заносит ее код в буфер клавиатуры, откуда он в дальнейшем считывается некоторой другой программой или операционной системой. На время такой обработки микропроцессор прекращает выполнение некоторой программы и переключается на так называемую *процедуру обработки прерывания*. После того, как данная процедура выполнит необходимые действия, прерванная программа продолжит выполнение с точки, где было приостановлено ее выполнение.

Некоторые ОС используют механизм прерываний не только для обслуживания внешних устройств, но и для предоставления своих услуг. Так, ОС MS DOS взаимодействует с системными и прикладными программами преимущественно через систему прерываний.

Прерывания могут быть *внешними* и *внутренними*. Внешние вызываются внешними по отношению к микропроцессору событиями, а внутренние возникают внутри него во время вычислительного процесса. Одной из причин возбуждения внутреннего прерывания является обработка машинной команды *int xx*, где *xx* — номер прерывания. Такие прерывания называются программными.

Далее рассмотрим различные функции прерываний базовой системы ввода/вывода.

3.2 Вывод на экран в текстовом режиме

3.2.1 Вывод средствами DOS. В предыдущих примерах использовался один из способов вывода текста на экран — вызов функции DOS 09h. Это далеко не единственный способ вывода текста. Все функции вызываются через прерывание 21h:

записать символ в STDOUT с проверкой на <Ctrl-Break>. Вход: *AH* = 02h; *DL* = ASCII-код символа. Обрабатываются управляющие символы: BEL (07h),

BS (08h), HT (09h), LF (0Ah), CR (0Dh) и др. Если в ходе работы этой функции была нажата комбинация клавиш <Ctrl-Break>, вызывается прерывание 23h (по умолчанию — выход из программы);

записать строку в STDOUT с проверкой на <Ctrl-Break>. Вход: *AH* = = 09h; *DS*:*DX* = адрес строки, заканчивающейся символом «$». Действие этой функции полностью аналогично действию функции 02h, но выводится не один символ, а целая строка.

3.2.2 Вывод средствами BIOS. Функции DOS вывода на экран позволяют перенаправлять вывод в файл, но не позволяют вывести текст в любую позицию экрана и изменить цвет текста. BIOS (базовая система ввода/вывода) — это набор программ, расположенных в постоянной памяти компьютера и обеспечивающих доступ к некоторым устройствам, в частности к видеоадаптеру. Все функции видеосервиса BIOS вызываются через прерывание 10h:

установить видеорежим. Вход: *AH* = 00; *AL* = номер режима в младших 7 битах. Вызов этой функции приводит к тому, что экран переводится в выбранный режим. Если старший бит *AL* не установлен в 1, экран очищается. Номера режимов: 0, 1 — 16 цветов, 40x25; 2, 3 — 16 цветов, 80x25; 7 — монохромный режим, 80x25;

установить положение курсора. Вход: *AH* = 02h; *BH* = номер страницы; *DH* = строка; *DL* = столбец. С помощью этой функции можно установить курсор в любую позицию экрана, и дальнейший вывод текста будет происходить из этой позиции. Символ в левой верхней позиции имеет координаты 0, 0;

считать положение и размер курсора. Вход: *AH* = 03h; *BH* = номер страницы (каждая страница использует собственный независимый курсор). Выход: *DH*, *DL* = строка и столбец текущей позиции курсора; *CH*, *CL* = первая и последняя строки курсора;

считать символ и атрибут символа в текущей позиции курсора. Вход: *AH* = 08h; *BH* = номер страницы. Выход: *AH* = атрибуты символа, изображенные на рисунке 3.1; *AL* = ASCII-код символа;

вывести символ с заданным атрибутом на экран. Вход: *AH* = 09h; *BH* = номер страницы; *AL* = ASCII-код символа; *BL* = атрибут символа; *CX* = число повторений символа. С помощью этой функции можно вывести на экран любой символ, включая даже символы CR и LF, которые обычно интерпретируются как конец строки;

вывести строку символов с заданными атрибутами. Вход: *AH* = 13h; *AL* = режим вывода: бит 0 — переместить курсор в конец строки после вывода; бит 1 — строка содержит не только символы, но и атрибуты, так что каждый символ описывается двумя байтами: ASCII-код и атрибут; *CX* = длина строки (только число символов); *BL* = атрибут, если строка содержит только символы; *DH*, *DL* = строка и столбец, начиная с которых будет выводиться строка; *ES*:*BP* = адрес начала строки в памяти. Функция выводит на экран строку символов, интерпретируя управляющие символы CR, LF, BS и BEL.

Мигание	Цвет фона			Яркие символы	Цвет символа		
7	6	5	4	3	2	1	0

Рисунок 3.1 — Атрибуты символа

В приведенной ниже таблице 3.1 перечислены двоичные значения цветов символов и фона и их названия.

Таблица 3.1 — Значения цветов символа и фона

Значение	Обычный цвет (цвет фона)	Яркий цвет
000b	черный	темно-серый
001b	синий	светло-синий
010b	зеленый	светло-зеленый
011b	голубой	светло-голубой
100b	красный	светло-красный
101b	пурпурный	светло-пурпурный
110b	коричневый	желтый
111b	светло-серый	белый

3.3 Ввод с клавиатуры

3.3.1 Ввод средствами DOS. Как и в случае вывода на экран, DOS предоставляет набор функций для чтения данных с клавиатуры, которые используют стандартное устройство ввода STDIN. Все функции вызываются через прерывание 21h:

считать строку символов из STDIN в буфер. Вход: *AH* = 0Ah, *DS*:*DX* = = адрес буфера. Выход: буфер содержит введенную строку.

Для вызова этой функции надо подготовить буфер, первый байт которого содержит максимальное число символов для ввода (1 — 254). При наборе строки обрабатываются клавиши: <Esc> — ввод сначала, <F3> — восстанавливает подсказку для ввода, <F5> — запоминает текущую строку как подсказку, <Backspace> — стирает предыдущий символ, <Ctrl-C>/<Ctrl-Break> — вызывается прерывание 23h. После нажатия клавиши Enter строка (включая последний символ CR = 0Dh) записывается в буфер начиная с третьего байта. Во второй байт записывается длина фактически введенной строки без учета последнего CR;

считать символ из STDIN с эхом, ожиданием и проверкой на <Ctrl-Break>. Вход: *AH* = 01h. Выход: *AL* = ASCII-код символа или 0. Если *AL* = 0, второй вызов этой функции возвратит в *AL* расширенный ASCII-код символа.

При чтении с помощью этой функции введенный символ автоматически немедленно отображается на экране. При нажатии <Ctrl-Break> выполняется команда INT 23h. Если нажата клавиша, не соответствующая какому-либо

символу (стрелки, функциональные клавиши Ins, Del и т.д.), то в *AL* возвращается 0 и функцию надо вызвать еще раз, чтобы получить расширенный ASCII-код.

Существуют варианты этой функции — 08h, 07h и 06h, в которых код символа возвращается по такому же принципу.

Кроме перечисленных функций могут потребоваться и некоторые служебные функции DOS для работы с клавиатурой:

проверить состояние клавиатуры. Вход: *AH* = 0Bh. Выход: *AL* = 0, если не была нажата клавиша, *AL* = 0FFh, если была нажата клавиша. Эту функцию удобно использовать перед функциями 01, 07 и 08, чтобы не ждать нажатия клавиши;

очистить буфер и считать символ. Вход: *AH* = 0Ch, *AL* = номер функции DOS (01, 06, 07, 08, 0Ah). Выход: зависит от вызванной функции.

3.3.2 Ввод средствами BIOS. Так же, как и для вывода на экран, BIOS предоставляет больше возможностей по сравнению с DOS для считывания данных и управления клавиатурой. Например, функциями DOS нельзя определить нажатие комбинаций клавиш типа <Ctrl-Alt-Enter> или нажатие двух клавиш <Shift> одновременно, DOS не может определить момент отпускания нажатой клавиши, и, наконец, в DOS нет функции, помещающей символ в буфер клавиатуры, как если бы его ввел пользователь. Все это можно осуществить, используя различные функции прерывания 16h:

чтение символа с ожиданием. Вход: *AH* = 10h. Выход: *AL* = ASCII-код символа, 0 или префикс скан-кода, *AH* = скан-код нажатой клавиши или расширенный ASCII-код.

Если нажатой клавише соответствует ASCII-символ, то в *AH* возвращается код этого символа, а в *AL* — скан-код клавиши. Если нажатой клавише соответствует расширенный ASCII-код, в *AL* возвращается префикс скан-кода или 0, а в *AH* — расширенный ASCII-код;

проверка символа. Вход: *AH* = 11h. Выход: ZF = 1 — буфер пуст, ZF = = 0 — в буфере присутствует символ, в этом случае: *AL* = ASCII-код символа, 0 или префикс скан-кода, *AH* = скан-код нажатой клавиши или расширенный ASCII-код;

поместить символ в буфер клавиатуры. Вход: *AH* = 05h, *CH* = скан-код, *CL* = ASCII-код. Выход: *AL* = 00h, если операция выполнена успешно, *AL* = 01h, если буфер клавиатуры переполнен. Например, следующая программа при запуске из DOS вызывает команду DIR:

```
;команда DIR выполняется сразу после завершения программы
.model tiny
.code
   org    100h
start:
   xor    si, si     ;счетчик
```

```
again:
    mov    cl, dir[si]   ;очередной символ команды
    xor    ch, ch
    mov    ah, 5
    int    16h           ;заносим в буфер клавиатуры
    inc    si            ;двигаемся к следующему символу
    cmp    si, len       ;повторяем, пока не достигли
    jnz    again         ;конца строки
    int    20h
dir db 'dir', 0dh
len =  $ - dir
end start
```

3.4 Работа с файлами

3.4.1 Открытие, закрытие, удаление файла. Прежде чем использовать файл в программе, его необходимо открыть, если он не существует — его нужно создать. Общий принцип работы с файлами следующий: открытие файла, какие-то действия над ним (чтение или запись), закрытие файла. Работа с файлами осуществляется через прерывание 21h:

создание или открытие файла. Вход: *AX* = 6C00h; *BX* = флаги, перечисленные в таблице 3.2; *CX* = атрибуты создаваемого (и только) файла (значения бит: 8–15 = 0 — резерв; 7 = 1 — общий файл в системе Novell Netware; 6 = 0 — резерв; 5 = 1 — бит архивации; 4 = 0 — зарезервирован (каталог), должен быть равен 0; 3 = 0 — игнорируется; 2 = 1 — системный файл; 1 = 1 — скрытый файл; 0 = 1 — только чтение); *DL* = действия, если файл существует или не существует, перечисленные в таблице 3.3; *DH* = 00h — резерв; *DS*:*SI* — адрес строки с ASCIZ-именем файла. Выход: CF = 0 — успешное выполнение функции: *AX* = дескриптор файла, CX = состояние (0 — файл открыт; 1 — файл создан и открыт; 2 — файл открыт без сохранения содержимого существующего файла); CF = 1 — *AX* = код ошибки;

закрытие файла не является обязательным, так как функция 4ch, которая завершает выполнение программы, в числе прочих действий выполняет и закрытие всех файлов. Вход: *AH* = 3Eh; *BX* = дескриптор файла, полученный при его открытии. Выход: CF = 0 — *AX* = не определен, CF = 1 — *AX* = код ошибки: 6 — недопустимый дескриптор файла.

Во время закрытия файла выполняются все незаконченные операции записи на диск в элементе каталога, соответствующего файлу, модифицируются различные поля (например, поля времени и даты устанавливаются в значения текущего времени и даты);

удаление файла. Вход: *AH* = 41h; *DS*:*DX* — ASCIZ-имя файла (можно использовать символы группирования «*» и «?»); *CL* = атрибуты удаляемого файла. Выход: CF = 0 — *AX* = не определен; CF = 1 — *AX* = код ошибки: 2 — файл не найден; 3 — нет такого пути; 5 — в доступе отказано. Функция не позволяет удалять файлы с атрибутом «только для чтения».

Таблица 3.2 — Флаги в регистре *BX* при создании или открытии файла

Биты	Назначение	Возможные значения
0–2	Указание режима доступа	000 — только чтение
		001 — только запись
		010 — чтение и запись
3	Резерв	0
4–6	Указание режима разделения	000 — режим совместимости
		001 — запрещение чтения и записи другим программам
		010 — запрещение записи другим программам
		011 — запрещение чтения другим программам
		100 — запрещение чтения и записи другим программам
7	Указание возможности наследования	0 — файл может наследоваться дочерними процессами
		1 — файл принадлежит только текущему процессу и не наследуется дочерними процессами
13	Указание способа обработки ошибок	1 — использовать обычный обработчик ошибок (int 24h)
		0 — использовать функцию 59h int 21h
14	Настройка буферизации	1 — использовать стандартную буферизацию
		0 — отменить стандартную буферизацию

Таблица 3.3 — Действия при существовании или несуществовании файла

Состояние	Биты	Возможные значения
Файл существует	0–3	0000 — вернуть ошибку
		0001 — открыть файл
		0010 — открыть файл без сохранения существующего
Файл не существует	4–7	0000 — вернуть ошибку
		0001 — открыть файл
		0010 — создать и открыть файл

3.4.2 Чтение, запись, позиционирование в файле. Функции чтения/записи можно использовать не только с дескрипторами заранее открытых файлов, но и с дескрипторами стандартных устройств. Эти дескрипторы имеют постоянные значения и доступны в любой момент: 0 — клавиатура (STDIN), 1 и 2 — экран (STDOUT и STDERR), 3 — последовательный порт COM1, 4 — параллельный порт LPT1. Все эти функции вызываются через прерывание 21h:

установка текущей файловой позиции. Чтение и запись в файле производятся с текущей файловой позиции, на которую указывает *файловый указатель*. Функция 42h MS DOS предоставляет гибкие возможности, как для начального,

так и для текущего позиционирования файлового указателя для последующей операции ввода- вывода.

Вход: *AH* = 42h; *BX* = дескриптор файла; *AL* = начальное положение в файле, относительно которого производится операция чтения/записи (00h — беззнаковое смещение от начала файла; 01h — смещение со знаком от текущей позиции в файле; 02h — смещение со знаком от конца файла); *CX*:*DX* = смещение новой позиции в файле относительно начальной. Выход: CF = 0 — *DX*:*AX* = = значение новой позиции в байтах относительно начала файла; CF = 1 — *AX* = = код ошибки: 1 — неверное значение в *AL*; 6 — недопустимый дескриптор файла.

Значение в *CX*:*DX*, позиционирующее указатель, в некоторых случаях может указывать за пределы файла. При этом выделяются два случая:

значение в *CX*:*DX* указывает на позицию перед началом файла — в этом случае последующая операция чтения/записи будет выполнена с ошибкой;

значение в *CX*:*DX* указывает на позицию за концом файла — в этом случае последующая операция записи приведет к расширению файла в соответствии со значением в *CX*:*DX*.

Примеры использования функции 42h приведены при рассмотрении функций чтения/записи:

запись в файл или устройство производится функцией 40h с текущей позиции файлового указателя. Вход: *AH* = 40h; *BX* = дескриптор файла; *CX* = количество байт для записи; *DS*:*DX* — указатель на область, из которой записываются данные. Выход: CF = 0 — *AX* = число действительно записанных байт в файл или устройство; CF = 1 — *AX* = код ошибки: 5 — в доступе отказано; 6 — недопустимый дескриптор.

Если при вызове функции 40h регистр *CX* равен нулю, то данные в файл не записываются, и он усекается или расширяется до текущей позиции файлового указателя. Если *CX* не равно нулю, то данные в файл записываются, начиная с текущей позиции файлового указателя. Операция записи также продвигает файловый указатель на число действительно записанных байт;

чтение из файла или устройства в область памяти осуществляется функцией 3Fh. Вход: *AH* = 3Fh; *BX* = дескриптор файла; *CX* = количество байт для чтения; *DS*:*DX* — указатель на область памяти, в которую помещаются прочитанные байты. Выход: CF = 0 — *AX* = число действительно прочитанных байт из файла; CF = 1 — *AX* = код ошибки: 5 — в доступе отказано; 6 — недопустимый дескриптор.

Чтение данных производится начиная с текущей позиции в файле, которая после успешного чтения смещается на значение, равное количеству прочитанных байт. Если в качестве файла используется стандартная консоль (клавиатура), то чтение производится до первого символа CR (carriage return) с кодом 0Dh, соответствующего нажатию клавиши Enter. Это, кстати, еще один способ ввода данных с клавиатуры в программу. Кроме символов введенной строки в ее конец помещаются символы 0Dh и 0Ah. Это надо учитывать при задании размера буфера для ввода.

Ниже приведен пример программы, в которой последовательно выполняются следующие действия: с клавиатуры вводится имя файла, открывается

файл, определяется его размер, содержимое файла читается в буфер, содержимое буфера выводится на экран:

```
.model small
.stack
.data
handle     dw  0       ;дескриптор файла
filename   db  20 dup(0) ;имя файла
p_fname        dd  filename  ;указатель на имя файла
buff       db  2048 dup(0)   ;буфер размером 2 Кбайт
p_buff     dd  buff       ;указатель на буфер
fsize      dw  0       ;длина файла
.code
start:
    mov ax, @data
    mov ds, ax     ;в ds — адрес сегмента данных
;читаем имя файла
    mov ah, 3fh        ;функция чтения
    mov bx, 0      ;дескриптор файла
    mov cx, 15     ;размер файла
    lds dx, p_fname   ;адрес буфера
    int 21h        ;читаем
    sub ax, 2      ;пропускаем символы 0dh и 0ah
    mov si, ax     ;ax содержит количество символов
    mov filename[si], 0  ;записываем в конец строки 0
;открываем или создаем файл
    xor cx, cx     ;атрибуты файла - обычный файл
    mov bx, 2      ;доступ для чтения-записи
    mov dx, 1      ;файл существует — открыть его,
                   ;иначе — вернуть ошибку
    lds si, p_fname   ;указатель на имя файла
    mov ah, 6ch        ;номер функции DOS
    int 21h        ;открываем файл
    jnc m1         ;если файл существовал, то переход
    mov dx,10h     ;создать файл
    mov ah,6ch     ;номер функции DOS
    int 21h        ;создаем файл
    jc  exit           ;выход, в случае ошибки
m1:                ;файл открыт успешно
    mov handle, ax;сохраним дескриптор файла
;определяем размер файла
    mov ah, 42h
    mov bx, handle
    mov al, 2      ;устанавливаем файловый указатель
    xor cx, cx     ;в конец файла
    xor dx, dx
    int 21h        ;в ax - размер файла (если только
```

```
    mov fsize, ax ;его длина не более 65535 байт)
    mov ah, 42h
    mov bx, handle
    mov al, 0     ;возвращаем файловый указатель
    xor cx, cx    ;на прежнее место (в начало)
    xor dx, dx
    int 21h
;читаем весь файл в буфер
    mov ah, 3fh       ;функция чтения
    mov bx, handle;дескриптор файла
    mov cx, fsize ;размер файла
    lds dx, p_buff;адрес буфера
    int 21h       ;читаем
    jc exit           ;если неудача - выход
    mov fsize, ax ;количество прочитанных байт
;вывод содержимого буфера на экран
    mov ah, 40h       ;записываем в
    mov bx, 1     ;STDOUT
    mov cx, fsize ;fsize байт
    lds dx, p_buff;из буфера buff
    int 21h
;закрываем файл
    mov ah, 3eh
    mov bx, handle
    int 21h
exit:             ;выход из программы
    mov ah, 4ch
    int 21h
end start
```

4 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить технологию создания EXE- и COM-программ;

б) набрать в любом текстовом редакторе примеры программ, приведенных в разделах 1 и 2. Получить EXE- и COM-модули;

в) изучить функции ввода/вывода DOS и BIOS. Получить индивидуальное задание у преподавателя. Выполнить индивидуальное задание, модифицировав одну из предыдущих программ;

г) изучить функции для работы с файлами. Получить индивидуальное задание у преподавателя. Модифицировать программу в соответствии с индивидуальным заданием.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

Лабораторная работа № 3

ИСПОЛЬЗОВАНИЕ ЦЕПОЧЕЧНЫХ КОМАНД

Цели работы: изучить принцип работы цепочечных команд и научиться использовать их в различных прикладных задачах.

1 Цепочечные команды

Цепочечные команды позволяют проводить действия над блоками памяти, представляющими собой последовательности элементов размером байт, слово, двойное слово. Содержимое этих блоков для микропроцессора не имеет никакого значения. Это могут быть символы, числа и все что угодно. Главное, чтобы размерность элементов совпадала с одной из перечисленных и эти элементы находились в соседних ячейках памяти.

Каждая цепочечная команда реализуется в микропроцессоре тремя командами, в зависимости от размера элемента, на что указывает последний символ в мнемонике команды (байт — *b*, слово — *w*, двойное слово — *d*). Особенность всех цепочечных команд в том, что они кроме обработки текущего элемента цепочки осуществляют еще и автоматическое продвижение к следующему элементу данной цепочки.

1.1 Префиксы повторений

Логически к цепочечным командам нужно отнести так называемые префиксы повторения, которые имеют свои мнемонические обозначения: *rep*, *repe* или *repz*, *repne* или *repnz*.

Они указываются перед нужной цепочечной командой, заставляя ее выполняться в цикле. Цепочечная команда без префикса выполняется один раз. Отличие приведенных префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: по состоянию регистра *ECX/CX* или по флагу нуля *ZF*.

Префикс *rep* (*repeat*) используется с командами, реализующими операции пересылки и сохранения элементов цепочек. Префикс *rep* заставляет данные команды выполняться в цикле, пока содержимое в *ECX/CX* не станет равным 0. При этом каждый раз содержимое *ECX/CX* автоматически уменьшается на единицу.

Префиксы *repe* и *repz* (*repeat while equal or zero*) являются абсолютными синонимами. Они заставляют цепочечную команду выполняться до тех пор, пока содержимое *ECX/CX* не равно нулю и флаг *ZF* равен 1. При этом каждый раз

содержимое *ECX/CX* автоматически уменьшается на единицу. Благодаря возможности анализа флага *ZF* наиболее эффективно эти префиксы можно использовать с командами сравнения и сканирования для поиска отличающихся элементов цепочек.

Префиксы *repne* и *repnz* (*repeat while not equal or zero*) также являются абсолютными синонимами, они заставляют цепочечную команду циклически выполняться до тех пор, пока содержимое *ECX/CX* не равно нулю и флаг *ZF* равен 0. Данные префиксы также можно использовать с командами сравнения и сканирования, но для поиска совпадающих элементов цепочек.

1.2 Адресация цепочек

Следующий важный момент, связанный с цепочечными командами, заключается в особенностях формирования физических адресов цепочки-источника и цепочки-приемника. Цепочка-источник может находиться в текущем сегменте данных, определяемом регистром *DS*. Цепочка-приемник должна быть в дополнительном сегменте данных, адресуемом сегментным регистром *ES*. Вторые части адресов — смещения цепочек — также находятся в строго определенных местах: для цепочки-источника это регистр *ESI/SI* (индексный регистр источника), для цепочки-приемника это регистр *EDI/DI* (индексный регистр приемника). Таким образом, полные физические адреса для операндов цепочечных команд следующие: адрес источника — пара *DS:ESI/SI*, адрес приемника — пара *ES:EDI/DI*.

1.3 Направление обработки цепочек

Обработка цепочек может происходить в двух направлениях: от начала цепочки к ее концу (в направлении возрастания адресов), от конца цепочки к началу (в направлении убывания адресов).

Цепочечные команды сами выполняют модификацию регистров, адресующих операнды, обеспечивая тем самым автоматическое продвижение по цепочке. Количество байт, на которые эта модификация осуществляется, определяется кодом команды, а знак этой модификации определяется значением флага направления *DF* (*Direction Flag*) в регистре *EFLAGS/FLAGS*:

если *DF* = 0, то значение индексных регистров *ESI/SI* и *EDI/DI* будет автоматически увеличиваться (операция инкремента) цепочечными командами, т.е. обработка будет осуществляться в направлении возрастания адресов;

если *DF* = 1, то значение индексных регистров *ESI/SI* и *EDI/DI* будет автоматически уменьшаться (операция декремента) цепочечными командами, т.е. обработка будет идти в направлении убывания адресов.

Состоянием флага *DF* можно управлять с помощью двух команд, не имеющих операндов: *cld* (*Clear Direction Flag*) — очистить флаг направления

(сбрасывает флаг направления *DF* в 0), *std* (*Set Direction Flag*) — установить флаг направления (устанавливает флаг *DF* в 1).

1.4 Пересылка цепочек

Команды, реализующие эту операцию, производят копирование элементов из одной области памяти (цепочки) в другую. Размер элемента определяется применяемой командой. Ассемблер предоставляет программисту три команды:

movsb — переслать цепочку байт;

movsw — переслать цепочку слов;

movsd — переслать цепочку двойных слов.

Каждая из команд пересылает только один элемент цепочки. Если перед командой указать префикс *rep*, то одной командой можно переслать до 64 Кбайт данных (если размер адреса 16 бит) или до 4 Гбайт данных (если размер адреса 32 бита). Число пересылаемых элементов должно быть загружено в регистр *ECX/CX*.

Фрагмент программы, выполняющей пересылку символов из одной цепочки в другую:

```
.data
...
source db 'Тестируемая цепочка', '$';цепочка-источник
str_len   dw $ - source      ;длина строки source
dest      db 20 dup (' ')        ;цепочка-приемник
...
.code
   ...
   mov   ax, ds
   mov   es, ax
   cld      ;обработка цепочки от начала к концу
   lea   si, source;в si адрес цепочки-источника
   lea   di, dest      ;в di адрес цепочки-приёмника
   mov   cx, str_len   ;счетчик повторений
   rep   movsb      ;пересылка цепочки
   lea   dx, dest
   mov   ah, 09h       ;вывод на экран цепочки dest
   int   21h
   ...
```

1.5 Сравнение цепочек

Команды, реализующие эту операцию, производят сравнение элементов цепочки-источника с элементами цепочки-приемника. Ассемблер предоставляет три команды сравнения цепочек, работающие с разными размерами элементов цепочки:

cmpsb — сравнить цепочку байт;

cmpsw — сравнить цепочку слов;

cmpsd — сравнить цепочку двойных слов.

Алгоритм работы команд сравнения заключается в последовательном выполнении операции вычитания над очередными элементами обеих цепочек. Принцип выполнения вычитания аналогичен команде сравнения *cmp* — результат никуда не записывается, но при этом модифицируются флаги *ZF*, *SF* и *OF*. С командами сравнения можно использовать префиксы повторений:

repe/repz — если необходимо организовать сравнение до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое *ECX/CX* стало равно 0) или в цепочках встретились разные элементы (флаг *ZF* стал равен 0);

repne/repnz — если нужно проводить сравнение до тех пор, пока не будет достигнут конец цепочки (содержимое *ECX/CX* стало равно 0) или в цепочках не встретятся одинаковые элементы (флаг *ZF* стал равен 1).

Таким образом, выбрав подходящий префикс, удобно использовать команды сравнения для поиска одинаковых или различающихся элементов цепочек.

После выхода из цикла в соответствующих индексных регистрах будут содержаться адреса элементов, следующих за теми, которые послужили причиной выхода из цикла. Поэтому, чтобы определить местоположение совпавших или не совпавших элементов в цепочках, необходимо скорректировать содержимое индексных регистров, увеличив либо уменьшив их содержимое на размер элемента цепочки.

В качестве примера рассмотрим фрагмент программы, которая сравнивает две цепочки, находящиеся в одном сегменте:

```
.data
match  db  0ah, 0dh, 'Цепочки совпадают.', '$'
failed db  0ah, 0dh, 'Цепочки не совпадают', '$'
;исследуемые цепочки
string1   db  '0123456789', 0ah, 0dh,'$'
string2   db  '0123406789', '$'
.code
   ...
   mov    ax, ds
   mov    es, ax     ;настройка es на ds
;вывод на экран исходных цепочек string1 и string2
```

```
    mov    ah, 09h
    lea    dx, string1
    int    21h
    lea    dx, string2
    int    21h
    cld
    lea    si, string1   ;в si адрес string1
    lea    di, string2   ;в di адрес string2
    mov    cx, 10    ;длина цепочки для repe
;сравнение цепочек (пока элементы цепочек равны)
;выход при обнаружении несовпавшего элемента
cycl:
    repe      cmpsb
    jne    not_match ;если не равны — переход на
                     ;not_match
equal:               ;иначе, если совпадают, то
    mov    ah, 09h       ;вывод сообщения
    lea    dx, match
    int    21h
    jmp    exit          ;выход
not_match:           ;не совпали
    mov    ah, 09h       ;вывод сообщения
    lea    dx, failed
    int    21h
;теперь, чтобы обработать несовпавший элемент в
;цепочке, необходимо уменьшить значения регистров si и di
    dec    si
    dec    di
;сейчас в ds:si и es:di адреса несовпавших элементов

;здесь вставить код по обработке несовпавшего элемента

;после этого продолжить поиск в цепочке:
    inc    si
    inc    di
    jmp    cycl
exit:            ;выход
    ...
```

1.6 Сканирование цепочек

Команды, реализующие эту операцию, производят поиск элемента в цепочке. Искомое значение предварительно должно быть помещено в регистр *AL/AX/EAX* (выбор регистра зависит от размера элементов цепочки).

Система команд микропроцессора предоставляет программисту три команды сканирования цепочки:

scasb — сканировать цепочку байт;

scasw — сканировать цепочку слов;

scasd — сканировать цепочку двойных слов.

Принцип поиска тот же, что и в командах сравнения, т.е. последовательное выполнение вычитания (*аккумулятор — элемент_цепочки*). В зависимости от результата вычитания производится установка флагов, при этом сами операнды не изменяются. С командами сканирования удобно использовать префиксы:

repe или *repz* — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое *ECX/CX* стало равно 0) или в цепочке встретился элемент, отличный от элемента в регистре *AL/AX/EAX*;

repne или *repnz* — если нужно организовать поиск до тех пор, пока не будет выполнено одно из двух условий: достигнут конец цепочки (содержимое *ECX/CX* стало равно 0) или в цепочке встретился элемент, совпадающий с элементом в регистре *AL/AX/EAX*.

Таким образом, команда сканирования с префиксом *repe/repz* позволяет найти элемент цепочки, отличающийся по значению от заданного в аккумуляторе, а с префиксом *repne/repnz* — совпадающий по значению с элементом в аккумуляторе.

В качестве примера рассмотрим фрагмент программы, который производит поиск символа в строке:

```
.data
;тексты сообщений
fnd    db 0ah, 0dh, 'Символ найден!', '$'
nochar db 0ah, 0dh, 'Символ не найден.', '$'
;цепочка для поиска
string db 'Поиск символа в этой цепочке.'
       db 0ah, 0dh, '$'
.code
    ...
    mov    ax, ds
    mov    es, ax     ;настройка es на ds
    mov    ah, 09h
    lea    dx, string
    int    21h        ;вывод сообщения string
```

```
    mov    al, 'a'       ;символ для поиска –
                         ;'a'(кириллица)
    cld                  ;сброс флага df
    lea    di, string;загрузка в di смещения цепочки
    mov    cx, 29        ;для repne — длина цепочки
;поиск в цепочке (пока искомый символ и символ в цепочке
не совпадут), выход при первом совпадении
    repne  scasb
    je     found  ;если равны — переход на обработку,
failed:                  ;иначе выполняем некоторые действия
;вывод сообщения о том, что символ не найден
    mov    ah, 09h
    lea    dx, nochar
    int    21h           ;вывод сообщения nochar
    jmp    exit             ;на выход
found:                   ;совпали
    mov    ah, 09h
    lea    dx, fnd
    int    21h           ;вывод сообщения fnd
;теперь, чтобы узнать место, где совпал элемент в
;цепочке, необходимо уменьшить значение в регистре di и
;вставить нужный обработчик
    dec    di
;обработчик
    ...
exit:          ;выход
    ...
```

1.7 Загрузка элемента цепочки в аккумулятор

Эта операция позволяет извлечь элемент цепочки и поместить его в регистр-аккумулятор *AL/AX/EAX*. Эту операцию удобно использовать вместе с поиском (сканированием) с тем, чтобы, найдя нужный элемент, извлечь его (например, для изменения). Размер извлекаемого элемента определяется применяемой командой. Программист может использовать три команды загрузки элемента цепочки в аккумулятор:

lodsb — загрузить байт из цепочки в регистр *AL*;

lodsw — загрузить слово из цепочки в регистр *AX*;

lodsd — загрузить двойное слово из цепочки в регистр *EAX*.

Работа команды заключается в том, чтобы извлечь элемент из цепочки по адресу, соответствующему содержимому пары регистров *DS:ESI/SI*, и поместить его в регистр *EAX/AX/AL*. Эту команду удобно использовать после команды сканирования, локализующей местоположение искомого элемента в цепоч-

ке. Префиксы повторений в этой команде могут и не понадобиться — все зависит от логики программы.

1.8 Перенос элемента из аккумулятора в цепочку

Данная операция позволяет сохранить значение из регистра-аккумулятора в элементе цепочки. Эту операцию удобно использовать вместе с операцией поиска (сканирования) и загрузки с тем, чтобы, найдя нужный элемент, извлечь его в регистр и записать на его место новое значение. Ассемблер предоставляет программисту три команды сохранения элемента цепочки из регистра-аккумулятора:

stosb — сохранить байт из регистра *AL* в цепочке;
stosw — сохранить слово из регистра *AX* в цепочке;
stosd — сохранить двойное слово из регистра *EAX* в цепочке.

Работа команды заключается в том, что она пересылает элемент из аккумулятора (регистра *EAX/AX/AL*) в элемент цепочки по адресу, соответствующему содержимому пары регистров *ES:EDI/DI*. Префиксы повторений в этой команде могут и не понадобиться — все зависит от логики программы. Например, если использовать префикс *rep*, то можно применить команду для инициализации области памяти некоторым фиксированным значением.

2 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить команды ассемблера для работы с цепочками;
б) изучить типы префиксов, используемых в цепочечных командах;
в) получить индивидуальное задание у преподавателя.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

Лабораторная работа № 4

СОВМЕСТНОЕ ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ АССЕМБЛЕР И C/C++. ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ В ОС WINDOWS

Цели работы: изучить технологию совместного программирования на языках C/C++ и ассемблер; научиться создавать модули на языке ассемблер, используемые в программах на C/C++.

1 Процедуры в программах на ассемблере

Процедуры на языке ассемблер определяются с помощью директивы *proc*, имеющей следующий синтаксис:

```
name_proc proc [language] [uses items,] [arguments]
```

где *name_proc* — имя процедуры; *language* — модификатор языка (см. директиву *public* в лаб. раб. № 1); *items* — список регистров, которые процедура не должна изменять; *arguments* — список параметров процедуры.

1.1 Сохраняемые регистры

Если указано предложение *uses* со списком регистров, то в начале процедуры будут вставлены команды сохранения этих регистров в стеке, а в конце — команды извлечения из стека. Например, если имеется процедура

```
any_proc proc uses si di
    ...; код процедуры
    ret
endp
```

то в процессе трансляции автоматически будет сгенерирован следующий код:

```
any_proc proc
    push   si
    push   di
    ...; код процедуры
    pop di
    pop si
    ret
endp
```

1.2 Параметры процедур

Параметры процедур описываются следующим образом:

```
arg1:type, [arg2:type, [arg3:type, ... [argN:type]]]
```

где *arg1, arg2, ..., argN* — имена параметров; *type* — тип параметра (*byte* — байт, *word* — слово, *dword* — двойное слово, *fword* — 6 байт, *qword* — учетверенное слово, *tbyte* — 10 байт, *near* — ближняя метка, *far* — дальняя метка).

Например:

```
any_proc proc uses si di, string:word, char:byte
    ... ; код процедуры
    ret
endp
```

1.3 Локальные переменные

Для распределения области под автоматические переменные в программе на ассемблере используется специальная директива *local*. Например:

```
local count:word, white:byte, file_cnt:word=size
```

определяет автоматические переменные *count*, *white* и *file_cnt*. Фактически эти переменные представляют собой метки, приравненные соответственно к *[EBP/BP-2]*, *[EBP/BP-4]*, *[EBP/BP-6]*. *Size* — это общее количество байт, необходимое для хранения автоматических переменных.

В общем виде директива *local* имеет следующий вид:

```
local name_1[[count][:type]][, name_2[[count][:type]]][=size]
```

где *name_n* — имена переменных; *count* — указывает, что переменная является массивом из *count* элементов; *type* — тип переменной или элементов массива (см. п. 1.2); *size* — метка (необязательное поле), значение которой будет равно размеру памяти в байтах, занимаемой локальными переменными.

Очень удобное свойство директивы *local* заключается в том, что метки, как для автоматических переменных, так и для общего размера автоматических переменных, ограничены по своей области определения процедурой, в которой они используются, что позволяет свободно давать автоматическим переменным в различных процедурах одинаковые имена.

2 Вызов ассемблерных функций из программ на C/C++

2.1 Организация интерфейса C/C++ и ассемблер

2.1.1 Соглашения об именах. Для компоновки модулей C/C++ и ассемблера необходимо, чтобы модули ассемблера соблюдали соглашения об именах, принятые в C/C++. Все имена в ассемблерных модулях, которые будут использоваться в модулях на C/C++, должны начинаться с символа «_». Кроме этого, необходимо учитывать, что по умолчанию ассемблер не различает регистр символов. Для того чтобы ассемблер различал регистр, необходимо транслировать ассемблерный модуль с ключом */ml* или */mx*.

2.1.2 Модели памяти. Для того чтобы та или иная ассемблерная функция могла быть вызвана из C/C++, она должна использовать ту же модель памяти, что и программа на C/C++, а также совместимый с C/C++ кодовый сегмент и сегмент данных. Аналогичным образом для того чтобы можно было из программы на C/C++ обращаться к данным, определенным в ассемблерном модуле, ассемблерная часть программы должна следовать соглашениям о наименованиях сегмента данных, принятым в языке C/C++.

2.1.3 Сохранение регистров. Что касается C/C++, то вызываемые из него ассемблерные функции могут делать все, что угодно, при условии, что они обеспечат сохранность следующих регистров: *EBP/BP*, *ESP/SP*, *CS*, *DS*, *SS*. Поскольку состояние этих регистров во время выполнения ассемблерной функции может меняться, то при возврате в вызывающую программу они должны иметь в точности то же самое значение, что и при входе в ассемблерную функцию. Регистры *EAX/AX*, *EBX/BX*, *ECX/CX*, *EDX/DX*, *ES* и регистр *EFLAGS/FLAGS* могут меняться любым образом.

Регистры *ESI/SI* и *EDI/DI* используются в C/C++ для хранения регистровых переменных. Поэтому, если программа, имеющая такие переменные, вызывает функцию на ассемблере, то вызываемая функция не должна менять содержимое этих регистров. В противном случае сохранение регистров *ESI/SI* и *EDI/DI* необязательно.

Между типами данных в C/C++ и ассемблере существует соответствие, представленное в таблице 2.1.

Таблица 2.1 — Соответствие типов данных языка C/C++ и ассемблер

Типы в C/C++	В программах для DOS	В программах для Win32
1	2	3
unsigned char	byte	byte
char	byte	byte
unsigned short	word	word
short	word	word
enum	word	dword

Окончание таблицы 2.1

1	2	3
unsigned int	word	dword
int	word	dword
unsigned long	dword	dword
long	dword	dword
float	dword	dword
double	qword	qword
long double	tbyte	tbyte
near *	word	dword
far *	dword	dword

2.2 Вызов функций

2.2.1 Передача параметров. C/C++ передает функциям параметры через стек. Перед вызовом функции C/C++ сначала помещает параметры, которые требуется передать этой функции, в стек, начиная с самого правого параметра и заканчивая левым. Например, вызов функции в C/C++

```
...
Test(i, j, k);
...
```

после компиляции принимает вид

```
...
push      _k
push      _j
push      _i
call      _Test
add    sp, 6
...
```

из чего следует, что самый правый параметр *k* помещается в стек первым, за ним *j* и, наконец, *i*.

После возврата из функции параметры, помещённые в стек, всё ещё находятся там, но никак не используются, поэтому непосредственно после каждого вызова функции C/C++ возвращает указатель стека назад, на значение, в котором он находился до помещения в стек параметров, и тем самым параметры уничтожаются. В предыдущем примере три параметра, по два байта каждый, занимают суммарно 6 байт стековой памяти, поэтому C/C++ прибавит к указателю стека число 6, обеспечив тем самым уничтожение параметров

после вызова *Test* (в C/C++ за уничтожение параметров в стеке отвечает вызывающая программа).

Ассемблерные функции обращаются к параметрам, переданным через стек относительно регистра *EBP/BP*. Например, имеется функция *Test*, находящая сумму из трёх передаваемых ей параметров:

```
.model small
.code
public _Test
_Test  proc
    push      bp
    mov    bp, sp
    mov    ax, [bp+4];принять параметр 1
    add    ax, [bp+6];прибавить параметр 2
    add    ax, [bp+8];прибавить параметр 3
    pop    bp
    ret
endp
end _Test
```

Test принимает параметры, переданные программой на C/C++ через стек относительно *BP* (*BP* адресует стековый сегмент).

На рисунке 2.1,а изображен стек программы сразу после входа в функцию *Test* (до выполнения первой команды). На рисунке 2.1,б показан этот же стек, но после выполнения команд

```
    push      bp
    mov    bp, sp
```

функции Test.

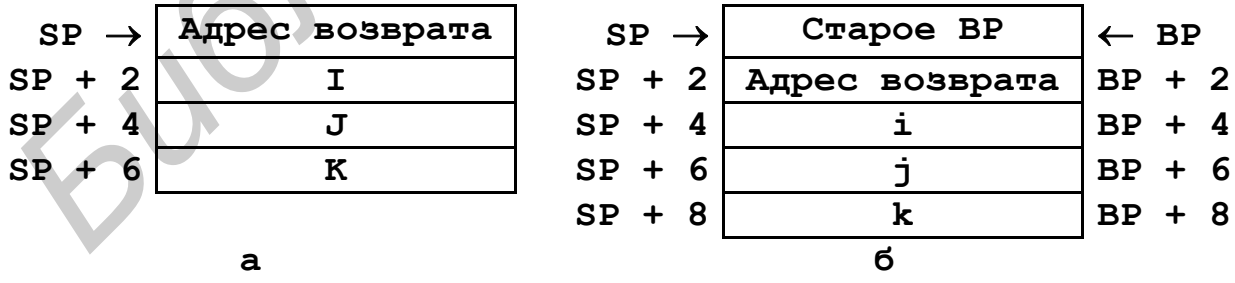

Рисунок 2.1 — Состояние стека в функции *Test*

2.2.2 Возврат значений. Вызываемая из C/C++ ассемблерная функция может возвращать значения так же, как и функция на C/C++. Образ возвращаемых из функции значений представлен в таблице 2.2.

Таблица 2.2 — Способы возврата значений

Тип возвращаемого значения	Для 16-разрядных приложений DOS	Для 32-разрядных приложений Win32
unsigned char	AX	EAX
char	AX	EAX
enum	AX	EAX
unsigned short	AX	EAX
short	AX	EAX
unsigned int	AX	EAX
int	AX	EAX
unsigned long	DX:AX	EAX
long	DX:AX	EAX
float	Регистр ST(0) стека FPU	Регистр ST(0) стека FPU
double	Регистр ST(0) стека FPU	Регистр ST(0) стека FPU
long double	Регистр ST(0) стека FPU	Регистр ST(0) стека FPU
near *	AX	EAX
far *	DX:AX	EAX

Следующий пример демонстрирует возврат значения функцией на ассемблере. Представленная в примере функция *FindLC* возвращает указатель на последний символ переданной ей строки:

```
// Программа на C
#include <stdio.h>
extern "C" char * FindLC(char *);

void main(void)
{
  char * TestString = "Кончается на У";

  printf("Строка \"%s\"заканчивается символом \'%c\'.\n",
        TestString, *FindLC(TestString));
}

; Программа на ассемблере
.model small
.code
public _FindLC
_FindLC proc
    push  bp
    mov   bp, sp
    push  di
    cld
    push  ds
```

```
    pop   es
    mov   di, [bp+4] ; установить es:di как указатель на
                     ; начало переданной строки
    mov   al, 0      ; ищется оканчивающий строку ноль
    mov   cx, 0ffffh ; поиск выполняется в 64 Кбайтах -1
    repnz scasb      ; поиск нуля
    dec   di         ; возврат указателя к нулю
    dec   di     ; возврат указателя к последнему символу
    mov   ax, di ; возврат ближнего указателя через ax
    pop   di
    pop   bp
    ret
_FindLC endp
end _FindLC
```

Конечный результат, а именно ближний указатель на последний символ в переданной строке возвращается в вызывающий модуль через *AX*.

2.3 Компоновка отдельных модулей в загрузочный файл

После создания исходных текстов программ на C/C++ и ассемблере с именами *name_c.cpp* и *name_a.asm* соответственно для получения загрузочного модуля в среде Borland C++ 3.1 необходимо выполнить одно из следующих действий:

а) создать проект, в который добавить созданные файлы, затем выполнить команду *Make*;

б) из командной строки выполнить

```
bcc -ms -Iinclude_path -Llib_path name_c.cpp name_a.asm
```

где *include_path* — путь к папке *include* с заголовочными файлами C/C++, а *lib_path* — путь к папке *lib*.

3 Программирование на ассемблере в ОС Windows

3.1 Ассемблер и Win32

Программирование на ассемблере под Win32 воспринимается весьма не однозначно. Считается, что написание приложений слишком сложно для применения ассемблера.

В отличие от программирования под DOS, где программы, написанные на языках высокого уровня, были мало похожи на свои аналоги, написанные на ассемблере, приложения под Win32 имеют гораздо больше общего. В первую очередь это связано с тем, что обращение к сервису операционной системы в

Windows осуществляется посредством вызова функций, а не прерываний, что было характерно для DOS. Здесь нет передачи параметров в регистрах при обращении к сервисным функциям и соответственно нет множества результирующих значений, возвращаемых в регистрах общего назначения и регистре флагов. Следовательно, проще запомнить и использовать протоколы вызова функций системного сервиса. С другой стороны, в Win32 нельзя непосредственно работать на аппаратном уровне, как в DOS. Вообще написание программ под Win32 стало значительно проще, и это обусловлено следующими факторами:

— гибкая система адресации к памяти — возможность обращаться к памяти через любой регистр общего назначения;

— доступность больших объемов виртуальной памяти;

— развитый сервис операционной системы, обилие функций, облегчающих разработку приложений;

— многообразие и доступность средств создания интерфейса с пользователем (диалоги, меню и т.п.).

Современный ассемблер, к которому относится и TASM 5.0 фирмы Borland International Inc., в свою очередь, развивал средства, которые ранее были характерны только для языков высокого уровня. К таким средствам можно отнести макроопределение вызова процедур, возможность введения шаблонов процедур (описание прототипов) и даже объектно-ориентированные расширения.

Все эти факторы позволяют рассматривать ассемблер как самостоятельный инструмент для написания приложений на платформе Win32.

3.2 Вызов функций Win32 API в программах на ассемблере

Функции Win32 API находятся в системных DLL-библиотеках. Поэтому их вызов осуществляется с учетом некоторых правил:

— все используемые функции Win32 API должны быть объявлены как внешние с помощью директивы *extrn*;

— программа должна компоноваться с библиотекой импорта *import32.lib*, в которой находится информация о местонахождении функций Win32 API;

— параметры функциям передаются через стек в обратном порядке;

— очистку стека от параметров осуществляет сама функция (используется соглашение *stdcall*).

Например, функцию Win32 API

```
int MessageBox(
    HWND hWnd,          //дескриптор окна владельца
    LPCTSTR lpText,     //адрес строки сообщения
    LPCTSTR lpCaption, //адрес строки заголовка сообщения
    UINT uType          //стиль окна
);
```

можно использовать в программе на ассемблере следующим образом:

```
.386          ;используем команды процессора 386
.model flat   ;модель памяти, аналогичная tiny, но с
          ;размером сегмента в 4 Гбайт

;указание tlink32, что программа должна компоноваться с
;библиотекой импорта import32.lib
includelibimport32.lib

;объявление внешних функций
extrn  MessageBoxA :proc
extrn  ExitProcess :proc
MB_OK  =   0

.data
szMessage db  'Это сообщение.', 0
szTitle       db  'Заголовок сообщения', 0

.code
start:
    xor    ebx, ebx
    push       MB_OK
    push       offset szTitle
    push       offset szMessage
    push       ebx
    call       MessageBoxA
    push       ebx
    call       ExitProcess   ;завершение программы
end start
```

Кроме этого, функции Win32 API можно объявлять в программах на ассемблере более удобным способом с помощью директивы *procdesc*. Использование этой директивы (а также ее преимущества) можно увидеть на следующем примере:

```
.386          ;используем команды процессора 386
.model flat, stdcall;используем для всех функций
                  ;соглашение stdcall (см. выше)
;указание tlink32, что программа должна компоноваться с
;библиотекой импорта import32.lib
includelibimport32.lib
;объявление внешних функций
MessageBoxA procdesc :dword, :dword, :dword, :dword
ExitProcess procdesc :dword
MB_OK  =   0
```

```
.data
szMessage db  'Это сообщение.', 0
szTitle       db  'Заголовок сообщения', 0

.code
start:
    xor    ebx, ebx
    call       MessageBoxA, ebx, offset szMessage, \
               offset szTitle, MB_OK
    call       ExitProcess, ebx ;завершение программы
end start
```

Как видно из примера, функция объявляется с указанием количества и типов параметров (обычно все параметры функций 32-разрядные), а вызов ее осуществляется в стиле языка C/C++.

3.3 Использование заголовочных файлов Win32

При программировании в Win32 часто приходится использовать множество констант, структур и типов данных, специфичных для Win32. Все они определяются в заголовочных файлах. Для их использования в программах на C/C++ обычно осуществляется подключение файла <*windows.h*>:

```
#include <windows.h>
```

Для программ на ассемблере также создаются заголовочные файлы. В поставку Turbo Assembler 5.0 входят файлы с определениями всех констант, структур и типов данных, используемых в Win32. Для их использования достаточно в программе указать директиву include с именем подключаемого файла. Например:

```
include windows.inc
```

4 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить технологию создания программ с совместным использованием C/C++ и ассемблера;

б) изучить принцип вызова процедур на ассемблере из программ на языке C/C++;

в) изучить особенности написания программ на ассемблере для Win32;

г) получить индивидуальное задание у преподавателя.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

Лабораторная работа № 5

ПРОГРАММИРОВАНИЕ МАТЕМАТИЧЕСКОГО СОПРОЦЕССОРА

Цели работы: изучить программную модель математического сопроцессора; научиться использовать команды математического сопроцессора.

1 Архитектура сопроцессора

С точки зрения программиста, сопроцессор представляет собой совокупность регистров, изображенную на рисунке 1.1, каждый из которых имеет свое функциональное назначение.

Рисунок 1.1 — Программная модель сопроцессора

В программной модели сопроцессора можно выделить три группы регистров:

— восемь регистров *R0 — R7*, составляющих основу программной модели сопроцессора — стек сопроцессора;

— три служебных регистра: регистр состояния сопроцессора *SWR* — отражает информацию о текущем состоянии сопроцессора; управляющий регистр сопроцессора *CWR* — управляет режимами работы сопроцессора; регистр слова тегов *TWR* — используется для контроля за состоянием каждого из регистров *R0 — R7*;

— два регистра указателей: данных — *DPR* и команд — *IPR*.

2 Форматы данных

2.1 Двоичные целые числа

Сопроцессор работает с тремя типами целых чисел, изображенными на рисунке 2.1. В таблице 2.1 представлены формат целых чисел, их размерность и диапазон значений.

Таблица 2.1 — Форматы целых чисел сопроцессора

Формат	Размер, бит	Диапазон значений
Целое слово	16	-32768…+32767
Короткое слово	32	$-2x10^9$…$+2x10^9$
Длинное целое	64	$-9x10^{18}$…$+9x10^{18}$

Следует помнить, что сопроцессор преобразовывает целые числа в их внутреннее представление в виде эквивалентных вещественных чисел расширенного формата, поэтому работа с целыми числами осуществляется сопроцессором неэффективно.

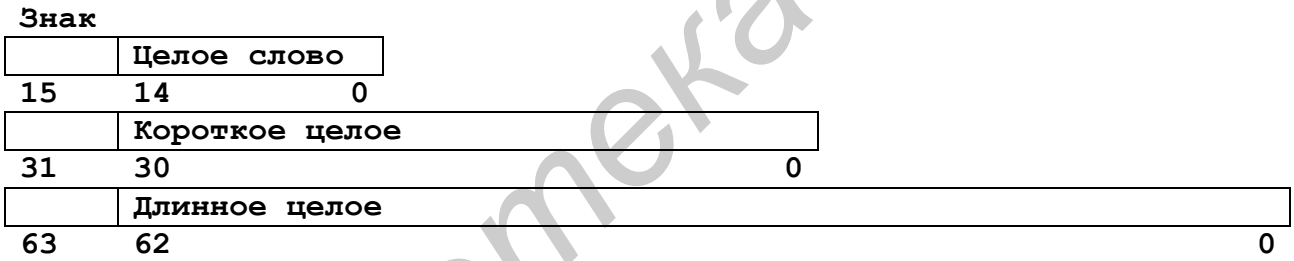

Рисунок 2.1 — Форматы целых чисел сопроцессора

В программе целые двоичные числа определяются обычным способом — с использованием директив *dw*, *dd* и *dq*.

2.2. Упакованные целые десятичные числа

Сопроцессор использует один формат упакованных десятичных чисел, изображенный на рисунке 2.2.

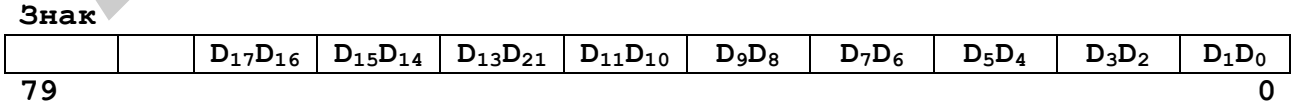

Рисунок 2.2 — Формат десятичного числа сопроцессора

Для описания упакованного десятичного числа используется директива *dt*, которая позволяет описать 20 цифр в упакованном десятичном числе (по две

в каждом байте). Но из-за того, что максимальная длина упакованного десятичного числа в сопроцессоре составляет только 9 байт, в регистры *R0* — *R7* можно поместить только 18 упакованных десятичных цифр. Старший байт игнорируется. Самый старший бит этого байта используется для хранения знака числа.

Например, целое число 5365904 в формате упакованного десятичного числа может быть описано следующим образом:

```
ch_dt  dt  5365904
;в памяти: ch_dt=04 59 36 05 00 00 00 00 00 00
```

2.3. Вещественные числа

Основной тип данных, с которым работает сопроцессор, вещественный. Данные этого типа описываются тремя форматами (рисунок 2.3): коротким, длинным и расширенным.

Рисунок 2.3 — Форматы вещественных чисел

В таблице 2.2 показаны диапазоны значений характеристик и мантисс вещественных чисел.

Таблица 2.2 — Форматы вещественных чисел

Формат	Короткий	Длинный	Расширенный
Длина числа (бит)	32	64	80
Размерность мантиссы (бит)	24	53	64
Размерность характеристики	8	11	15
Диапазон значений	$10^{-38}…10^{+38}$	$10^{-308}…10^{+308}$	$10^{-4932}…10^{+4932}$

Короткое вещественное число длиной 32 разряда определяется директивой *dd*. При этом обязательным в записи числа является наличие десятичной точки, даже если оно не имеет дробной части. Для транслятора десятичная точка является указанием, что число нужно представить в виде числа с плавающей точкой в коротком формате. Это же касается длинного и расширенного форматов, определяемых директивами *dq* и *dt*. Другой способ задания вещественного числа директивами *dd*, *dq* и *dt* — экспоненциальная форма. Например:

```
;определение вещественных чисел короткого формата
sh_real_1 dd 45.56
sh_real_2 dd 37.12e7
sh_real_3 dd 0.915e-3
;определение вещественных чисел длинного формата
lg_real_1 dq 3.14159
lg_real_2 dq -2846.0
lg_real_3 dq -1.5e-2
;определение вещественных чисел расширенного формата
ex_real_1 dt 2.73
ex_real_2 dt 10.10e10
ex_real_3 dt -0.1e2
```

2.4 Специальные численные значения

Несмотря на большой диапазон вещественных значений, которые могут быть представлены в регистрах сопроцессора, есть бесконечное множество значений, которые находятся за рамками этого диапазона. Для того чтобы иметь возможность реагировать на вычислительные ситуации, в которых возникают такие значения, в сопроцессоре предусмотрены специальные комбинации бит, называемые специальными численными значениями.

2.4.1 Денормализованные вещественные числа — это числа, которые меньше минимального нормализованного числа для каждого вещественного формата.

Как известно, сопроцессор хранит числа в нормализованном виде. По мере приближения чисел к нулю сопроцессору все труднее «вытягивать» их значения к нормализованному виду, то есть к такому виду, чтобы первой значащей цифрой мантиссы была единица.

2.4.2 Нуль также относится к специальным численным значениям, это делается из-за того, что это значение особо выделяется среди корректных вещественных значений, формируемых как результат работы некоторой команды.

Значение истинного нуля может иметь знак (рисунок 2.4), что, впрочем, не влияет на его использование командами сопроцессора.

0	000 · · · · 000	000 · 000
79	78 64	63 0
1	000 · · · · 000	000 · 000
79	78 64	63 0

Рисунок 2.4 — Представление нуля в регистре стека сопроцессора

2.4.3 Бесконечность — формат регистра стека сопроцессора, содержащего значение «бесконечность», приведен на рисунке 2.5.

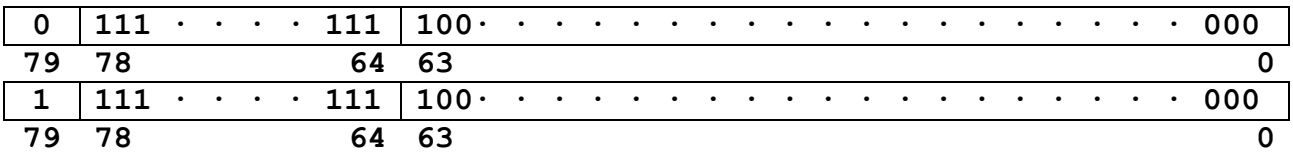

Рисунок 2.5 — Представление бесконечности

Значение «бесконечность» может иметь знак, при этом значения мантиссы и характеристики фиксированы. Именно в этом заключается отличие значения «бесконечность» от остальных специальных значений.

3 Система команд сопроцессора

3.1 Команды передачи данных

Группа команд передачи данных предназначена для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти. С помощью этих команд осуществляются все перемещения значений операндов в сопроцессор и из него.

В сопроцессоре имеются следующие команды передачи данных:

fld/fild источник — загрузка вещественного/целого числа из памяти в вершину стека сопроцессора;

fst/fist приемник — сохранение вещественного/целого числа из вершины стека сопроцессора в память. Сохранение числа в памяти не сопровождается выталкиванием его из стека, то есть текущая вершина стека сопроцессора не изменяется (поле *top* не изменяется);

fstp/fistp приемник — сохранение вещественного/целого числа из вершины стека сопроцессора в память. В отличие от предыдущей команды происходит выталкивание вещественного числа из стека после его сохранения в памяти. Команда изменяет поле *top*, увеличивая его на единицу. Вследствие этого вершиной стека становится следующий, больший по своему физическому номеру, регистр стека сопроцессора;

fxch st(i) — обмен значений между текущей вершиной стека и регистром стека сопроцессора *ST(i)*.

3.2 Команды загрузки констант

В математических вычислениях достаточно часто встречаются предопределенные константы. Сопроцессор хранит значения некоторых из них. Для каждой предопределенной константы существует своя специальная команда:

fldz — загрузка нуля в вершину стека сопроцессора;

`fld1` — загрузка единицы в вершину стека сопроцессора;

`fldpi` — загрузка числа π в вершину стека сопроцессора;

`fldl2t` — загрузка двоичного логарифма десяти в вершину стека сопроцессора;

`fldl2e` — загрузка двоичного логарифма *e* в вершину стека сопроцессора;

`fldlg2` — загрузка десятичного логарифма двух в вершину стека сопроцессора;

`fldln2` — загрузка натурального логарифма двух в вершину стека сопроцессора.

3.3 Команды сравнения данных

Команды данной группы выполняют сравнение значений числа, находящегося в вершине стека и операнда, указанного в команде:

`fcom/ficom операнд` — команда без операндов сравнивает два значения: одно находится в регистре *ST(0)*, другое — в регистре *ST(1)*. Если указан *операнд*, то сравнивается значение в регистре *ST(0)* стека сопроцессора со значением в памяти;

`fcomp/ficomp операнд` — команда сравнивает значение в вершине стека сопроцессора *ST(0)* со значением операнда, который находится в регистре или в памяти. После сравнения и установки бит *c3–c0* команда выталкивает значение из *ST(0)*. Длина целого операнда 16 или 32 бита, то есть целое слово и короткое целое;

`ftst` — команда не имеет операндов и сравнивает значения в *ST(0)* со значением 0.

В результате работы команд сравнения в регистре состояния *SWR* устанавливаются следующие флаги *c3*, *c2*, *c0*:

если операнды несравнимы, то *c3=1*, *c2=1*, *c0=1*;

если *операнд_2 > операнд_1* (*ST(1) > ST(0)*), то *c3=0*, *c2=0*, *c0=1*;

если *операнд_2 < операнд_1* (*ST(1) < ST(0)*), то *c3=0*, *c2=0*, *c0=0*;

если *операнд_2 = операнд_1* (*ST(1) = ST(0)*), то *c3=1*, *c2=0*, *c0=0*.

Для программирования реакции на результаты сравнения необходимо анализировать состояние этих бит. Сопроцессор не имеет для этого специальных команд, за исключением команды *fstsw*, которая позволяет записать содержимое регистра *SWR* в регистр *AX*. Затем командой *sahf* содержимое регистра *AH*, в котором находятся биты *c3*, *c2* и *c0*, записывается в младший байт регистра *EFLAGS/FLAGS*. Местоположение бит *c3*, *c2* и *c0* соответствует местоположению флагов *ZF*, *PF* и *CF*. Таким образом, после применения команды *sahf* становится возможным реагировать на результаты сравнения значений в сопроцессоре командами условного перехода основного процессора.

3.4 Арифметические команды

Команды сопроцессора, входящие в данную группу, реализуют четыре основные арифметические операции — сложение, вычитание, умножение и деление. Имеется также несколько дополнительных команд, предназначенных для повышения эффективности использования основных арифметических команд.

Целочисленные арифметические команды:

`fiadd источник` — команда складывает значения *ST(0)* и целочисленного источника, в качестве которого выступает 16- или 32-разрядный операнд в памяти. Результат сложения запоминается в регистре стека сопроцессора *ST(0)*;

`fisub источник` — команда вычитает значение целочисленного иточника из *ST(0)*. Результат вычитания запоминается в регистре стека сопроцессора *ST(0)*. В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти;

`fimul источник` — команда умножает значение целочисленного источника на содержимое *ST(0)*. Результат умножения запоминается в регистре стека сопроцессора *ST(0)*. В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти;

`fidiv источник` — команда делит содержимое *ST(0)* на значение целочисленного источника. Результат деления запоминается в регистре стека сопроцессора *ST(0)*. В качестве источника выступает 16- или 32-разрядный целочисленный операнд в памяти.

Вещественные арифметические команды:

`fadd` — команда складывает значения из регистров *ST(0)* и *ST(1)*. Результат сложения запоминается в регистре стека сопроцессора *ST(0)*;

`fadd слагаемое` — команда складывает значения *ST(0)* и *слагаемого*, представляющего собой адрес ячейки памяти. Результат сложения запоминается в регистре стека сопроцессора *ST(0)*;

`fadd st(i), st` — команда складывает значение в регистре стека сопроцессора *ST(i)* со значением в вершине стека *ST(0)*. Результат сложения запоминается в регистре *ST(i)*;

`faddp st(i), st` — команда производит сложение вещественных операндов аналогично команде *fadd st(i), st*. Последним действием команды является выталкивание значения из вершины стека сопроцессора *ST(0)*. Результат сложения остается в регистре *ST(i – 1)*;

`fsub` — команда вычитает из значения в регистре *ST(1)* значение регистра *ST(0)*. Результат вычитания запоминается в регистре стека сопроцессора *ST(1)*;

`fsub вычитаемое` — команда вычитает значение *вычитаемого* из значения в *ST(0)*. *Вычитаемое* представляет собой адрес ячейки памяти, содержащей допустимое вещественное число. Результат сложения запоминается в регистре стека сопроцессора *ST(0)*;

`fsub st(i), st` — команда вычитает значение в вершине стека *ST(0)* из значения в регистре стека сопроцессора *ST(i)*. Результат вычитания запоминается в регистре стека сопроцессора *ST(i)*;

`fsubp st(i), st` — команда вычитает вещественные операнды аналогично команде *fsubp st(i), st*. Последним действием команды является выталкивание значения из вершины стека сопроцессора *ST(0)*. Результат вычитания остается в регистре *ST(i-1)*;

`fmul` — команда (без операндов) умножает значение в *ST(1)* на содержимое регистра в *ST(0)*. Результат умножения запоминается в регистре стека сопроцессора *ST(1)*;

`fmul множитель` — команда умножает *множитель* на содержимое регистра стека *ST(0)*. Результат умножения запоминается в регистре стека сопроцессора *ST(0)*;

`fmul st(i), st` — команда умножает значение в произвольном регистре *ST(i)* на содержимое регистра стека *ST(0)*. Результат умножения запоминается в регистре стека сопроцессора *ST(i)*;

`fmulp st(i), st` — команда производит умножение подобно команде *fmul st(i), st*. Последним действием команды является выталкивание значения из вершины стека сопроцессора *ST(0)*. Результат умножения остается в регистре *ST(i – 1)*;

`fdiv` — команда (без операндов) делит содержимое регистра *ST(1)* на значение регистра сопроцессора *ST(0)*. Результат деления запоминается в регистре стека сопроцессора *ST(0)*;

`fdiv делитель` — команда делит содержимое регистра *ST(0)* на *делитель*. Результат деления запоминается в регистре стека сопроцессора *ST(0)*;

`fdiv st(i), st` — команда производит деление содержимого регистра *ST(i)* на значение регистра сопроцессора *ST(0)*. Результат деления запоминается в регистре стека сопроцессора *ST(i)*;

`fdivp st(i), st` — команда производит деление аналогично команде *fdiv st(i), st*. Последним действием команды является выталкивание значения из вершины стека сопроцессора *ST(0)*. Результат деления остается в регистре *ST(i – 1)*.

Дополнительные арифметические команды:

`fsqrt` — вычисление квадратного корня из значения, находящегося в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов. Результат вычисления помещается в регистр *ST(0)*. Следует отметить, что данная команда обладает определенными достоинствами. Во-первых, результат извлечения достаточно точен, во-вторых, скорость исполнения чуть больше скорости выполнения команды деления вещественных чисел *fdiv*;

`fabs` — вычисление модуля значения, находящегося в вершине стека сопроцессора — регистре *ST(0)*. Команда не имеет операндов. Результат вычисления помещается в регистр *ST(0)*;

`fchs` — изменение знака значения, находящегося в вершине стека сопроцессора — регистре *ST(0)*. Команда не имеет операндов. Результат вычисления помещается обратно в регистр *ST(0)*. Отличие команды *fchs* от команды *fabs* в том, что команда *fchs* только инвертирует знаковый разряд значения в регистре *ST(0)*, не меняя значения остальных бит. Команда вычисления модуля *fabs* при наличии отрицательного значения в регистре *ST(0)* наряду с инвертированием знакового ряда выполняет изменение остальных бит значения таким образом, чтобы в *ST(0)* получилось соответствующее положительное число;

`fxtract` — команда выделения порядка и мантиссы значения, находящегося в вершине стека сопроцессора — регистре *ST(0)*. Команда не имеет операндов. Результат выделения помещается в два регистра стека: мантисса — в *ST(0)*, а порядок — в *ST(1)*. При этом мантисса представляется вещественным числом с тем же знаком, что и у исходного числа, и порядком, равным нулю. Порядок, помещенный в *ST(1)*, представляется как истинный порядок, то есть без константы смещения, в виде вещественного числа со знаком и соответствует величине p формулы $A=(\pm M)*N^{\pm(p)}$;

`fscale` — команда умножает или делит значение, находящееся в вершине стека сопроцессора *ST(0)*, на степень двух. Значение показателя степени находится в регистре стека *ST(1)* и представляет собой целое число со знаком. Если показатель степени не является целым числом, то он округляется до целого в меньшую сторону. Команда не имеет операндов. Команда fscale не очищает регистр *ST(1)*. Результат помещается в регистр *ST(0)*;

`frndint` — команда округления до целого значения — округляет значение, находящееся в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов.

3.5 Команды трансцендентных функций

Сопроцессор имеет ряд команд, предназначенных для вычисления значений тригонометрических функций, таких, как синус, косинус, тангенс, арктангенс, а также значений логарифмических и показательных функций. Наличие этих команд значительно облегчает разработку вычислительных алгоритмов. В системе команд сопроцессора имеются следующие трансцендентные команды:

`fcos` — команда вычисляет косинус угла, находящегося в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов. Результат возвращается в регистр *ST(0)*;

`fsin` — команда вычисляет синус угла, находящегося в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов. Результат возвращается в регистр *ST(0)*;

`fsincos` — команда вычисляет синус и косинус угла, находящегося в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов. Результат возвращается в регистры *ST(0)* и *ST(1)*. При этом косинус помещается в *ST(0)*, а синус — в *ST(1)*;

`fptan` — команда вычисляет частичный тангенс угла в радианах, (в диапазоне $-2^{63}\ldots2^{63}$), находящегося в вершине стека сопроцессора *ST(0)*. Команда не имеет операндов. Результат возвращается в регистры *ST(0)* — значение единицы, и *ST(1)* — тангенс угла;

`fpatan` — команда вычисляет частичный арктангенс частного *x/y* (*x* находится в регистре *ST(0)*, *y* — в регистре *ST(1)*). Команда не имеет операндов. Результат возвращается в регистре *ST(0)*;

`fprem` — команда получения частичного остатка от деления. Исходные значения делимого и делителя размещаются в стеке: делимое — в *ST(0)*, делитель — в *ST(1)*. Делитель рассматривается как некоторый модуль, поэтому в результате работы команды получается остаток от деления по модулю. Но произойти это может не сразу, так как этот результат в общем случае достигается за несколько производимых подряд обращений к команде *fprem*. Это происходит, если значения операндов сильно различаются. Физическая работа команды заключается в реализации хорошо известного действия деления в столбик. При этом каждое промежуточное деление осуществляется отдельной командой *fprem*. Цикл, центральное место в котором занимает команда *fprem*, завершается, когда очередная полученная разность в *ST(0)* становится меньше значения модуля в *ST(1)*. Судить об этом можно по состоянию флага *c2* в регистре состояния *SWR*:

если *c2 = 0*, то работа команды *fprem* полностью завершена, так как разность *ST(0)* меньше значения модуля в *ST(1)*;

если *c2 = 1*, то необходимо продолжить выполнение команды *fprem*, так как разность *ST(0)* больше значения модуля в *ST(1)*;

`f2xm1` — команда вычисления значения функции $y = 2^{x-1}$. Исходное значение *x* размещается в вершине стека сопроцессора в регистре *ST(0)* и должно лежать в диапазоне $-1 \le x \le 1$. Результат *y* замещает *x* в регистре *ST(0)*. Эта команда может быть использована для вычисления различных показательных функций;

`fyl2x` — команда вычисления значения функции $z = y\log_2(x)$. Исходное значение *x* размещается в вершине стека сопроцессора, а исходное значение *y* — в регистре *ST(1)*. Значение *x* должно лежать в диапазоне $0 \le x \le +\infty$, а значение *y* — в диапазоне $-\infty \le y \le +\infty$. Перед тем, как осуществить запись результата *z* в вершину стека, команда *fyl2x* выталкивает значения *x* и *y* из стека и только после этого производит запись *z* в регистр *ST(0)*;

`fyl2xp1` — команда вычисления значения функции *z = ylog2(x+1)*. Исходное значение *x* размещается в вершине стека сопроцессора — регистре *ST(0)*, а исходное значение *y* — в регистре *ST(1)*.). Значение *x* должно лежать в

диапазоне $0 \le x \le 1-1/\sqrt{2}$, а значение y — в диапазоне $-\infty \le y \le +\infty$. Перед тем, как осуществить запись результата z в вершину стека, команда *fyl2xp1* выталкивает значения x и y из стека и только после этого производит запись z в регистр *ST(0)*.

3.6 Команды управления сопроцессором

Последняя группа команд предназначена для общего управления работой сопроцессора:

`wait/fwait` — команда ожидания, предназначена для синхронизации работы процессора и сопроцессора;

`finit` — команда инициализации сопроцессора. Данную команду используют перед первой командой сопроцессора в программе и других случаях, когда необходимо привести сопроцессор в начальное состояние;

`fstsw назначение` — команда сохранения содержимого регистра состояния *SWR* в ячейке памяти размером 2 байта или регистре *AX*;

`ffree st(i)` — команда освобождения регистра стека *ST(i)*. Команда записывает в поле регистра тегов, соответствующего регистру *ST(i)*, значение *11b*, что соответствует пустому регистру.

4 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить программную модель математического сопроцессора;

б) изучить систему команд математического сопроцессора;

в) получить индивидуальное задание у преподавателя.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

Лабораторная работа № 6

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ MMX

Цели работы: изучить технологию, программную модель MMX-расширения и систему команд MMX-расширения.

1 Технология MMX

Технология Intel MMX включает набор расширений к архитектуре Intel, которые разработаны, чтобы увеличить производительность средств мультимедиа и коммуникаций.

Эти расширения (которые включают новые регистры, типы данных и инструкции) объединены с моделью выполнения «одна инструкция, много данных» (SIMD). Расширение MMX предназначено для ускорения выполнения приложений типа «подвижное видео», комбинированной графики с видеообработкой изображений, звуковым синтезом, синтезом и сжатием речи, телефонией, конференц-связью и 2D- и 3D-графикой, которые типично используют алгоритмы с интенсивными вычислениями, чтобы исполнять повторяющиеся действия на больших множествах простых элементов данных.

Технология MMX определяет простую и гибкую модель программного обеспечения без нового режима или видимого состояния для операционной системы. Все существующее программное обеспечение будет работать правильно, без модификации, на процессорах архитектуры Intel, которые включают технологию MMX, даже в присутствии существующих и новых приложений, которые включают эту технологию.

Следующие разделы этой работы описывают основную окружающую среду программирования технологии MMX, включая набор регистров MMX, типы данных и набор инструкций.

2 Технология программирования MMX

Технология MMX обеспечивает следующие новые расширения к окружающей среде программирования архитектуры Intel:

восемь MMX-регистров (от *MM0* до *MM7*);

четыре MMX-типа данных (упакованные байты, упакованные слова, упакованные двойные слова и четверное слово (quadword));

систему команд MMX.

2.1 Регистры MMX

Набор регистров MMX состоит из восьми 64-разрядных регистров. MMX-команды обращаются к регистрам MMX непосредственно, используя имя регистра от *MM0* до *MM7*. Эти регистры могут использоваться только для выполнения вычислений над MMX-типами данных; они не могут использоваться для адресации памяти. Адресация операндов MMX-команды в памяти осуществляется, используя стандартные способы адресации и регистры *EAX*, *EBX*, *ECX*, *EDX*, *EBP*, *ESI*, *EDI* и *ESP*.

Хотя регистры MMX определены в архитектуре Intel как отдельные регистры, они являются псевдонимами младших 64-разрядных частей регистров в FPU стеке (*R0...R7*), как изображено на рисунке 2.1.

Рисунок 2.1 — Регистры MMX и регистры FPU

Когда регистры сопроцессора играют роль MMX-регистров, то доступными являются лишь их младшие 64 бита. К тому же при работе стека сопроцессора в режиме MMX-расширения он рассматривается не как стек, а как обычный регистровый массив с произвольным доступом. Регистровый стек сопроцессора не может одновременно использоваться и по своему прямому назначению, и как MMX-расширение. Забота о его разделении и корректной работе с ним ложится на программиста.

2.2 Типы данных MMX

Технология MMX определяет следующие новые 64-разрядные типы данных (рисунок 2.2):

упакованные байты — восемь байт, упакованные в одно 64-разрядное поле;

упакованные слова — четыре слова, упакованные в одно 64-разрядное поле;

упакованные двойные слова — два двойных слова, упакованные в одно 64-разрядное поле;

четверное слово — одно 64-разрядное поле.

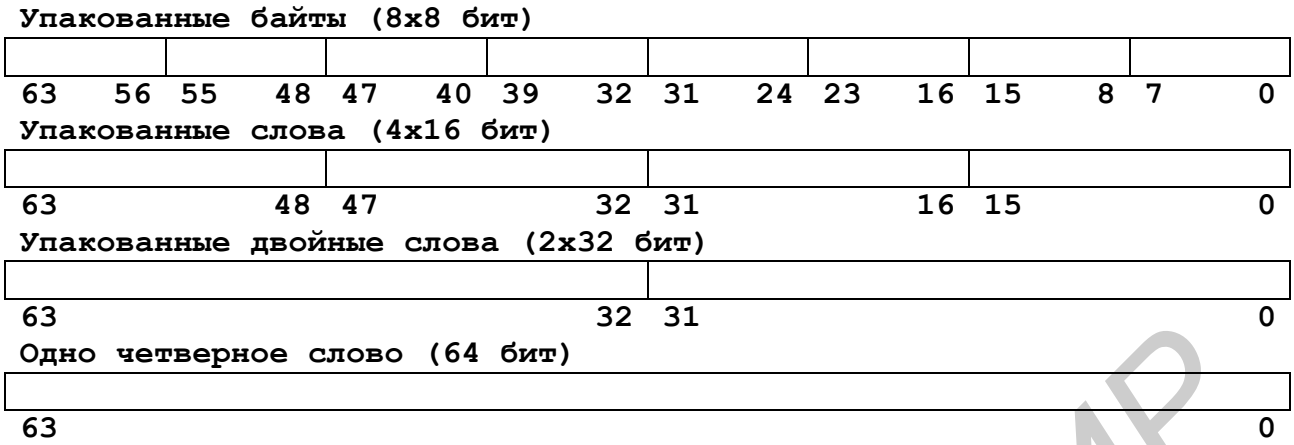

Рисунок 2.2 — Типы данных MMX

MMX-команды перемещают упакованные байты, упакованные слова, упакованные двойные слова и четверные слова в (из) память или в (из) универсальные регистры архитектуры Intel в 64-разрядных блоках. Однако при выполнении арифметических или логических операций над упакованными типами данных MMX-команды оперируют параллельно над индивидуальными байтами, словами или двойными словами, содержащимися в 64-разрядном MMX-регистре. При операциях над байтами, словами и двойными словами внутри упакованных типов данных MMX-команды оперируют как со знаковыми, так и беззнаковыми целыми байтами, словами, двойными словами.

2.3 Модель SIMD

MMX-технология использует методику «одиночная команда, множественные данные» (SIMD) для выполнения арифметических и логических операций над байтами, словами или двойными словами, упакованными в 64-разрядные регистры MMX. Например, команда *paddsb* складывает 8 знаковых байт исходного операнда с 8 знаковыми байтами в операнде адресата и сохраняет 8 результирующих байт в операнде адресата. Эта SIMD методика ускоряет эффективность выполнения программ, позволяя одну и ту же операцию выполнять параллельно на множестве элементов данных.

Модель выполнения SIMD, обеспечиваемая в MMX-технологии, удовлетворяет потребностям современных средств связи и графических приложений, которые часто используют сложные алгоритмы, в которых выполняются одни и те же операции над большим количеством данных. Например, большие звуковые данные представляются в 16-разрядных словах. Команды MMX могут обрабатывать сразу 4 из этих слов одновременно в одной команде. Видео- и графическая информация обычно представляются как пакетированные 8-разрядные байты. Одна MMX-команда может оперировать над 8 из этих байт одновременно.

2.4 Форматы данных в памяти

Когда упакованные байты, слова или двойные слова сохраняются в памяти, они сохраняются по последовательным адресам. Самый младший байт, слово или двойное слово сохраняются по самому младшему адресу, а старшие байты, слова или двойные слова — по более старшим адресам.

3 Система команд MMX

Система команд MMX состоит из 57 команд, сгруппированных в следующие категории: команды передачи данных, арифметические команды, команды сравнения, команды преобразования, логические команды, команды сдвига, команда *emms* — освободить состояние MMX.

При оперировании над упакованными данными внутри регистра MMX данные приводятся в соответствии с типом, определенным командой. Например, команда *paddb* (добавить упакованные байты) обрабатывает упакованные данные в регистре MMX как 8 упакованных байт, в то время как команда *paddw* (добавлять упакованные слова) обрабатывает упакованные данные как 4 упакованных слова.

3.1 Арифметика насыщенности и режим цикличности

Технология MMX поддерживает новую арифметическую возможность, известную как арифметика *насыщенности* (*saturated arithmetics*). Насыщенность лучше всего определить, противопоставляя ее режиму *цикличности*. В режиме цикличности результаты, которые *переполняются* или *антипереполняются*, усечены и возвращаются только самые младшие биты результата; т.е. перенос игнорируется. В режиме насыщенности результаты операции, которые переполняются или антипереполняются, отсекаются к границе для типа данных (таблица 3.1). Результат операции, которая превышает диапазон типа данных, насыщается к максимальному значению диапазона, а результат, который является меньше, чем диапазон типа данных, — к минимальному значению диапазона. Этот метод обработки переполнения и антипереполнения применяется во многих приложениях.

Например, когда результат превышает диапазон данных для знаковых байт, он обрезается до 7fh (0ffh для байт без знака). Если значение меньше диапазона, оно обрезается до 80h для знаковых байт (00h для байт без знака).

Таблица 3.1

Тип данных	Нижний предел		Верхний предел	
	шестнадцат.	десятичн.	шестнадцат.	десятичн.
Знаковый байт	80h	-128	7fh	127
Знаковое слово	8000h	-32768	7fffh	32767
Беззнаковый байт	00h	0	0ffh	255
Беззнаковое слово	0000h	0	0ffffh	65535

3.2 Операнды команд

Все команды MMX, за исключением *emms*, оперируют с двумя операндами: первый операнд — адресат, второй операнд — источник. Команда записывает результат в операнд-адресат.

Исходный операнд для всех MMX команд (за исключением команд передачи данных) может быть или в памяти, или в регистре MMX. Операнд адресата всегда в регистре MMX. Для команд передачи данных операндом источника и адресата может также быть целочисленный регистр (для команды *movd*) или память (для *movd* и *movq* команд).

3.3 Команды передачи данных

Команда *movd* передает 32-разрядные упакованные данные из памяти в регистры MMX и обратно или из целочисленных регистров процессора в регистры MMX и обратно. Команда *movq* передает 64-разрядные упакованные данные из памяти в регистры MMX и обратно или между регистрами MMX.

3.4 Арифметические команды

3.4.1 Упакованное сложение и вычитание. Команды *paddsb*, *paddsw* и *paddwd* (упакованное сложение) и *psubb*, *psubw* и *psubd* (упакованное вычитание) выполняют сложение или вычитание знаковых или беззнаковых элементов данных операнда источника операнда адресата в циклическом режиме. Эти команды поддерживают упакованные байты, упакованные слова и упакованные двойные слова.

Команды *paddsb* и *paddsw* (упакованное сложение с насыщенностью) и *psubsb* и *psubsw* (упакованное вычитание с насыщенностью) выполняют сложение или вычитание знаковых элементов данных операнда источника и операнда адресата и приводят результат к граничным значениям диапазона знакового типа данных. Эти команды поддерживают упакованные байты и упакованные слова.

Команды *paddusb* и *paddusw* (упакованное сложение без знака с насыщенностью) и *psubusb* и *psubusw* (упакованное вычитание без знака с насыщенностью) выполняют сложение или вычитание элементов данных без знака операнда источника и операнда адресата и приводят результат к граничным значениям диапазона типа данных без знака. Эти команды поддерживают упакованные байты и упакованные слова.

3.4.2 Упакованное умножение. Команды упакованного умножения выполняют четыре умножения на парах 16-разрядных знаковых операндов, производя 32-разрядные промежуточные результаты. Пользователи могут выбирать старшие или младшие части каждого 32-разрядного результата.

Команды *pmulhw* и *pmullw* умножают знаковые слова операндов источника и адресата и записывают старшую или младшую часть результата в операнд адресата.

3.4.3 Упакованное умножение/сложение. Команда *pmaddwd* (упакованное умножение и сложение) вычисляет произведение знаковых слов операндов адресата и источника. Четыре промежуточных 32-разрядных произведения суммируются в парах, чтобы произвести два 32-разрядных результата.

3.5 Команды сравнения

Команды *pcmpeqb*, *pcmpeqw*, *pcmpeqd* (упакованное сравнение на равенство) и *pcmpgtb*, *pcmpgtw*, *pcmpgtd* (упакованное сравнение на «больше») сравнивают соответствующие элементы данных в операндах источника и адресата на равенство или оценивают кто из них больше. Эти команды генерируют маску единиц или нулей, которые записываются в операнд адресата. Логические операции могут использовать маску, чтобы выбрать элементы. Это можно использовать, чтобы выполнить упакованную условную операцию пересылки без ветвления или набора команд ветвления. Никакие флаги не устанавливаются. Эти команды поддерживают упакованные байты, упакованные слова и упакованные двойные слова.

3.6 Команды преобразования

Команды преобразования преобразовывают элементы данных внутри упакованного типа данных.

Команды *packsswb* и *packssdw* (упакованный со знаковой насыщенностью) преобразовывают знаковые слова в знаковые байты или знаковые двойные слова в знаковые слова в режиме знаковой насыщенности.

Команда *packuswb* (упакованный насыщенностью без знака) преобразовывает знаковые слова в байты без знака в режиме насыщенности без знака.

Команды *punpckhbw*, *punpckhwd* и *punpckhdq* (распаковать старшие упакованные данные) и *punpcklbw*, *punpcklwd* и *punpckldq* (распаковать младшие упакованные данные) преобразовывают байты в слова, слова в двойные слова или двойные слова в четверное слово.

3.7 Логические команды

Команды *pand* (поразрядное логическое И), *pandn* (поразрядное логическое И-НЕ), *por* (поразрядное логическое ИЛИ) и *pxor* (поразрядное логическое исключающее ИЛИ) выполняют поразрядные логические операции над 64-разрядными данными.

3.8 Команды сдвига

Команды логического сдвига влево, логического сдвига вправо и арифметического сдвига право сдвигают каждый элемент на определенное число бит. Логические левые и правые сдвиги также дают возможность перемещать 64-разрядное поле как один блок, что помогает в преобразованиях типа данных и операциях выравнивания.

Команды *psllw*, *pslld* (упакованный логический сдвиг влево) и *psrlw*, *psrld* (упакованный логический сдвиг вправо) выполняют логический левый или правый сдвиг и заполняют пустые старшие или младшие битовые позиции нулями. Эти команды поддерживают упакованные слова, упакованные двойные слова и четверное слово.

Команды *psraw* и *psrad* (упакованный арифметический сдвиг вправо) выполняют арифметический сдвиг вправо, копируя знаковый разряд в пустые разрядные позиции на старшем конце операнда. Эти команды поддерживают упакованные слова и упакованные двойные слова.

3.9 Команда EMMS

Команда *emms* освобождает состояние MMX. Эта команда должна использоваться, чтобы очистить состояние MMX (чтобы освободить *tag*-слово регистров FPU) в конце MMX подпрограммы перед вызовом других подпрограмм, которые могут выполнять операции с плавающей точкой.

Наличие этой команды обязательно, если MMX-команды комбинируются с командами сопроцессора.

4 Порядок выполнения работы

Для выполнения данной работы необходимо следующее:

а) изучить программную модель MMX;

б) изучить систему команд MMX;

в) получить индивидуальное задание у преподавателя.

Работа оценивается по результатам выполнения индивидуального задания, а также по знанию теоретической части лабораторной работы.

ЛИТЕРАТУРА

1. Ахо А. В., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Учеб. пособие / Пер. с англ. — М.: Вильямс, 2000.
2. Бек Л. Введение в системное программирование: Пер. с англ. — М.: Мир, 1988.
3. Григорьев В. Л. Архитектура и программирование арифметического сопроцессора. — М.: Энергоатомиздат, 1991.
4. Гук М. Аппаратные средства IBM PC: Энциклопедия. — СПб.: Питер, 1999.
5. Гук М., Юров В. Процессоры Pentiun 4, Athlon и Duron. — СПб.: Питер, 2001.
6. Зубков С.В. Ассемблер для DOS, Windows и UNIX. — М.: ДМК Пресс, 2000.
7. Использование Turbo Assembler при разработке программ. — Киев: Диалектика, 1994.
8. Сван Т. Освоение Turbo Assembler. — Киев: Диалектика, 1996.
9. Юров В. Assembler. — СПб.: Питер, 2001.
10. Юров В. Assembler: Практикум. — СПб.: Питер, 2001.
11. Юров В. Assembler: Специальный справочник. — СПб.: Питер, 2001.

Св. план 2002, поз. 36

Учебное издание

Лепешинский Владимир Николаевич

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по курсу
МИКРОПРОЦЕССОРЫ И МИКРОКОМПЬЮТЕРЫ
для студентов специальности 53 01 02
«Автоматизированные системы обработки информации»
дневной формы обучения

Редактор Н.А. Бебель
Корректор Е.Н. Батурчик
Компьютерная верстка Т.В. Шестакова

Подписано в печать 15.11.2002. Формат 60x84 1/16.
Бумага офсетная. Печать ризографическая. Гарнитура «Таймс». Усл. печ. л. 4,1.
Уч.-изд. л. 3,5. Тираж 150 экз. Заказ 208

Издатель и полиграфическое исполнение:
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия ЛП № 156 от 05.02.2001.
Лицензия ЛВ № 509 от 03.08.2001.
220013, Минск, П. Бровки, 6.