САНКТ–ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

С. А. МОЛОДЯКОВ, А. В. ПЕТРОВ

АРХИТЕКТУРА ЭВМ
НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Санкт-Петербург
Издательство Политехнического университета
2019

УДК 004.42

ББК 32.97

Молодяков С. А., Петров А. В. **АРХИТЕКТУРА ЭВМ. НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ:** лабораторный практикум**.** С.А.Молодяков, А.В.Петров, - Изд-во Политехн. ун-та, 2019. – 81 с.

Лабораторный практикум предназначен для помощи студентам при проведении лабораторных работ по дисциплине «Архитектура ЭВМ». В лабораторном практикуме рассмотрены вопросы низкоуровневого программирования, разработки программ на языках Ассемблера и Си. Значительное внимание уделено использованию встроенного Ассемблера и его связи с функциями Си.

Учебное пособие предназначено для студентов, обучающихся по направлениям подготовки 09.03.04 «Программная инженерия» и 02.03.02 «Фундаментальная информатика и информационные технологии», изучающих дисциплины «Архитектура ЭВМ», «Программирование периферийных устройств» и др.

Печатается по решению редакционно-издательского совета Санкт-Петербургского политехнического университета Петра Великого.

Оглавление

Введение

В учебном пособии по лабораторному практикуму рассмотрены вопросы низкоуровневого программирования компьютеров и периферийных устройств. Изучение программирования микропроцессора Pentium проводится на пользовательском и системном уровнях.

В качестве инструмента при выполнении лабораторных работ предлагается использовать языки Ассемблера и Си. Основной средой разработки может быть Borland C или Visual C (Microsoft Studio). Можно использовать другие системы программирования, позволяющие иметь доступ к ресурсам компьютера на уровне физических адресов ячеек памяти и портов, а также к регистрам процессора.

Лабораторные работы построены на последовательном освоении и нарастании сложности программирования. Они начинаются с изучения кодов команд и способов адресации и заканчиваются написанием программ типа «Отладчик» или «Мультизадачный вывод».

Представленная информация и примеры программ дают возможность студентам самостоятельно подготовиться к лабораториям, изучить материал в большем объеме, чем предусматривают лабораторные работы.

Описанные лабораторные работы проводятся в первом семестре при изучении дисциплины «Архитектура ЭВМ» направлений подготовки «Программная инженерия» и «Фундаментальная информатика и информационные технологии».

Лабораторная работа №1.

Введение в низкоуровневое программирование.

Встроенный отладчик. Встроенный Ассемблер

В качестве инструмента при выполнении лабораторных работ предлагается использовать Borland C при желании можно взять Visual C (Microsoft Visual Studio). Среда разработки Borland C имеет встроенный ассемблер, встроенный отладчик IDE и экранный отладчик TD.

Дается шаблон программы, предлагается изучить использование средств Ассемблера в программе, а также отладку, как на уровне исходного Си-текста, так и на уровне отладчика.

Цель работы: 1. Знакомство с программными средствами для работы с аппаратурой.

2. Изучение программной модели процессора x86

Информация: Описание программной модели процессора. Организация памяти. Пример программы. Используйте описания работы элементов среды Borland C.

Задание на выполнение работы

1. Прочитайте текст программы и выделите логически связные части, поймите, что делают эти части. Разбирая текст, обратите внимание на то, какие функции Си используются и зачем. Делая это, научитесь пользоваться контекстным Help'ом. Познакомьтесь, как включать элементы встроенного Ассемблера в текст программы.

2. Скомпилируйте программу и убедитесь, что в ней отсутствуют ошибки.

3. Запустите программу, наблюдайте результат и убедитесь, что Вы понимаете, что происходит.

4. Научитесь, используя встроенный отладчик IDE:

- Выполнить программу до заданной точки останова. На уровне исходного Си-текста это можно сделать, используя пункты меню Run и Debug/Breakpoints.
- Проверить/изменить содержимое переменных после останова. Используйте пункт меню Debug/Evaluate/Modify, Debug/Inspect (горячая клавиша Alt/F4) либо Debug/Watches

- Измерить время выполнения заданной преподавателем команды так, как это сделано в программе LAB0 с командой
  MOV REG,MEM.

5. Средствами отладчика TD в окне CPU (пункт меню View/CPU) научитесь:

- - наблюдать/изменять содержимое регистров процессора (подумайте, какие регистры можно менять безболезненно (изменения в некоторых регистрах могут привести к фатальным для нормального выполнения программы результатам))
- - наблюдать/изменять ячейку памяти с известным физическим адресом:

(Прежде, чем писать куда-либо, хорошо было бы подумать: что это за адрес - адрес ОЗУ (не использует ли этот адрес еще какая-нибудь программа), адрес ПЗУ, существует ли физическое устройство, соответствующее данному адресу...

Варианты:

1. 0x46C 0x80000 0xF000:0xFFF0	2. 0x0040:0x6D 0x8000:0x0010 0xFFFFE	3. 0x41A 0x80210 0xFFFF:0x100

- - читать/писать в произвольный порт ввода/вывода (например: считать порт 0x40 несколько раз - попробуйте объяснить наблюдаемый результат; записать число 3 в 61 порт, потом быстро записать туда нуль);

6. **Определите в какие команды Ассемблера оттранслирован с языка СИ оператор цикла.** Напишите на уровне Ассемблера свой вариант цикла. Сравните, результаты представьте преподавателю.

Описание программной модели процессора.

Конкретный набор команд, форматы и способы адресации определяются структурой процессора и организацией памяти. Рассмотрим структуру микропроцессора i8086 - предка процессоров Pentium.

Программисту на уровне команд доступны четырнадцать регистров. Их удобно разбить на четыре группы: 1)Регистры данных, 2)адресные, 3)сегментные 4)указатель команд и регистр флажков(признаков).

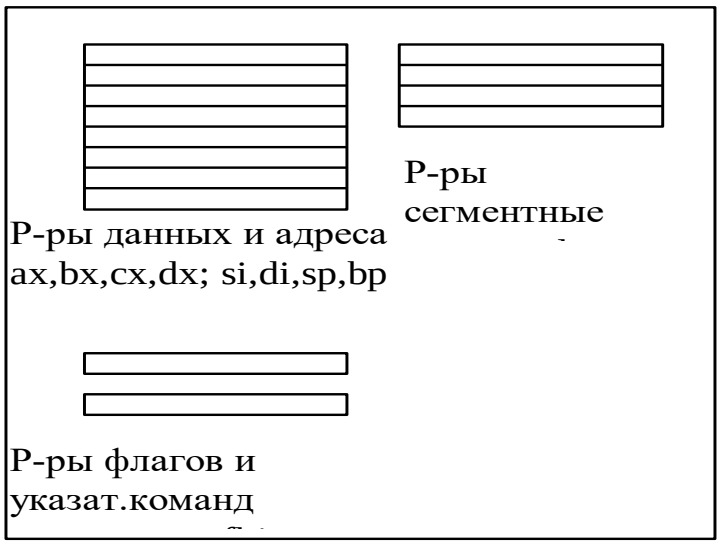
1) Регистры данных (в некоторых книгах их называют регистрами общего назначения). Операнды в этих регистрах могут быть как слова, так и

байты. Если операнд - байт, может быть указана любая половина регистра. Есть ряд команд, в которых функции отдельных регистров специализированы (см.табл.)

2) Указатели и индексные регистры (адресные регистры, используются для хранения 16-разрядных адресов).

3) Сегментные регистры (указывают начала четырех сегментов - участков по 64 К байт в 1М ОЗУ: сегмент команд CS, сегмент стека SS и два сегмента данных - DS и ES extra)

4) Указатель команд и регистр флажков



Функции регистров 8086

AX Аккумулятор Умножение, деление и ввод-вывод слов
AL Аккумулятор(мл) Умножение, деление и ввод-вывод байтов
AH Аккумулятор(ст) Умножение и деление байтов
BX Регистр базы Базовый регистр, может быть использован как адресный
CX Счетчик Операции с цепочками, циклы
CL Счетчик (мл) Динамические сдвиги и ротации
DX Данные Умножение и деление слов, косвенный ввод-вывод
SP Указатель стека Стековые операции
BP Указатель базы Базовый регистр
SI Индекс источника Операции с цепочками, индексный регистр
DI Индекс приемника Операции с цепочками,

Регистр флагов процессора 8086

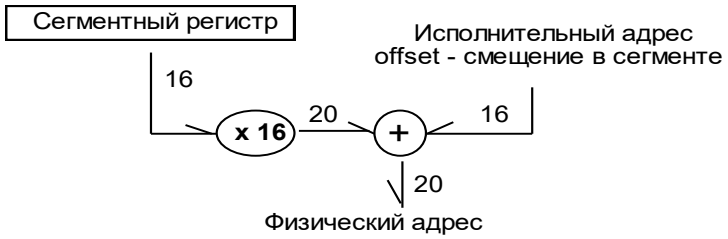
15				11	10	9	8	7						1	0
				OF	DF	IF	TF	SF	ZF	0	AF	0	PF	0	CF

CF (Carry Flag) - флаг переноса;
PF (Parity Flag) - флаг четности;
AF (Auxiliary Carry Flag) - флаг вспомогательного переноса;
ZF (Zero Flag) - флаг нуля;
SF (Sign Flag) - флаг знака;
TF (Trap Flag) - флаг прерывания для отладки
IF (Interrupt-Enable Flag) - флаг разрешения прерывания;
DF (Direction Flag) - флаг направления цепочечных команд
OF (Overflow Flag) - флаг переполнения.

Организация памяти.

Для формирования физических адресов используется механизм сегментации памяти. Необходимость сегментной организации памяти определяется недостаточным размером адресных регистров для адресации к памяти большей 64 Кбайт. Для организации адресации в процессоре используются четыре (в i80386 и выше шесть) 16-разрядных сегментных регистра, которые указывают на начала четырех сегментов памяти - участков по 64 К байт: сегмент команд CS, сегмент стека SS и два сегмента данных - DS и ES extra). Местоположение операнда в сегменте определяется смещением (offset).

Адресуемая память (адресное пространство) представляет собой область из 1М байт. Два смежных байта образуют слово. Адресом слова считается адрес младшего байта. Физический адрес памяти имеет длину 20 бит, однако, все обрабатываемые в регистрах процессора величины имеют длину 16 бит. Схема трансляции приведена на рисунке.



Как было уже сказано, пространство памяти 1 М доступно процессору через 4 "окна" (сегмента). Начальный адрес каждого сегмента содержится в одном из четырех сегментных регистров. Команды обращаются к байтам и словам в пределах сегментов, используя относительный (внутрисегментный) адрес. Таким образом, адрес операнда задается двумя компонентами: адресом сегмента и смещением. Адрес сквозной нумерации 0xb8000 задается в виде: **0xb800:0.**

Адресное пространство процессора Intel 8086 (реальный режим адресации)


```
//      Пример оформления программы

//                      Работа N 0
//   Измерение времени выполнения фрагмента программы
//   Студенты гр 000.0 Иванофф, Петрофф, & Сидорофф
//
#include <dos.h>
#include <bios.h>
#include <stdio.h>
#include <conio.h>
#define PortCan0   0x40

void beep(unsigned iTone,unsigned iDlit);  // Это прототип
функции
     // звукового сигнала, используемой при измерении времени

void main(void)
{
// Объявление переменных
long int lCnt=0;    //  Ячейка - счетчик повторений
int iA=0x1234;      //  Думми-ячейка, используемая в
исследуемой команде

/*********************************************\
/* Как из Си посмотреть содержимое байта с известным
физическим адресом */

// Если хотим напечатать содержимое байта с адресом 0046Ch
```

```
//объявляем far-указатель на переменную типа char  и
инициализируем
//этот указатель значением адреса, предварительно преобразовав
//его к типу char *
char far * pT=(char *)0x46C;                          // (1)
  printf ("\n Печатаем 10 раз значение байта с известным
адресом \n");
  for (int i=0; i<10; i++)
    printf (" \n  %d ", * pT);                        // (1)
  printf ("\n Для продолжения нажмите любую клавишу \n");
  getch();     // Программа ждет нажатия клавиши

/*****************************************\
\* Как из Си посмотреть содержимое порта  */

//Читаем содержимое порта с адресом 40 с помощью функции Си
   printf ("\n Читаем содержимое порта с адресом 40 с помощью
функции Си \n" );
//   Цикл повторяется каждые 0.5 с
   printf ("\n Для выхода из цикла - нажмите любую клавишу \n"
);

     while (bioskey(1)==0)  // пока не будет нажата любая
клавиша
      {
printf (" \n Порт40 = %d ", inp(PortCan0));      //(2)
//  С помощью TD посмотрите, во что превращается ф-ция inp()
// на уровне машинных команд
       delay(500);      //  Задержка на полсекунды (500 мс)
      }
     getch();  // Очищаем буфер клавиатуры
/************************************************\
  Примечания:                                   *
Функция printf (...) позволяет распечатать на экране значения
переменных,  *
а также произвольный текст.                          *
Функция bioskey(1) позволяет определить, нажата ли клавиша *
Функция inp(uPort) позволяет считать байт из порта Port
*
Функция outp(uPort,iValue) позволяет вывести величину iValue в
порт uPort  *
Функция delay(uTime) организует программную задержку на uTime
миллисекунд  *
Функция getch() считывает один символ из буфера клавиатуры.
*
  В данном случае это надо для очистки буфера клавиатуры
*
\************************************************/
```

```
// Снова читаем тот же порт с помощью программы на встроенном
ассемблере
   printf ("\n Читаем содержимое порта с адресом 40
ассемблером \n" );

  while (bioskey(1) == 0 )  //  Этот цикл будет повторяться,
                  // пока не нажмем клавишу

// Примеры использования встроенного ассемблера (3)
   {
   asm {  push ax   // Один способ записи на встроенном
ассемблере
       in al,0x40
     }

       unsigned char Tmm = _AL;  //  Эта команда эквивалентна
mov Tmm,al
             // !!  Убедитесь в этом с помощью TD

       asm    pop ax // Другой способ записи на встроенном
ассемблере
       delay (500);
       printf (" \n Порт40 = %d ", Tmm );
       // Если нажата клавиша - то выход
   }
     getch();
     printf ("\n Для продолжения - нажмите любую клавишу \n
");
     getch();

/*****************************************************
   Как посмотреть содержимое длинной (например)четырехбайтовой
 переменной с адреса 0046C с помощью средств Си
*/
long far * pTime=(long *)0x46C;   // Указатель на счетчик
тиков
 while (bioskey(1) == 0)
    {
    printf ("\n %ld",*pTime);
    delay(1000);
    }
    getch();
   // Читаем и печатаем содержимое двухбайтовой переменной
   // с адреса 0046C средствами встроенного ассемблера
 int Time;
 while (bioskey(1) == 0)
    {
     asm push ds       //  Сохраним на всякий случай регистры
     asm push si
```

```
                              //  Во встроенном ассемблере
     asm mov ax,40h     // можно записывать hex-константы так
...
     asm mov ds,ax
     asm mov si,0x6C    //  ... или так
     asm mov ax,[ds:si]
     asm mov Time,ax
     asm pop si         //  А теперь восстановим регистры
     asm pop ds         // (не перепутайте порядок !!!)

     printf ("\n %d",Time);
     delay(300);
     }

/*********************************************************
       Пример выполнения задания типа И
   Требуется измерить время выполнения заданной команды
   (в примере - команда mov reg,mem, а Вы спросите у
преподавателя
   какую команду Вам взять )

   Измерение времени выполнения фрагмента программы    */
 beep(400,200);  //  Сигнал отмечает начало интервала   (5)
      for ( lCnt=0; lCnt<1000000; lCnt++)
     {
a1:    asm { mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
           mov ax,iA
a2:        mov ax,iA }
     }
      beep(400,200); //Сигнал отмечает конец интервала (5)
}
// Функция подачи звукового сигнала заданной высоты и
длительности (5)
void beep(unsigned iTone,unsigned iDlit)
{  sound(iTone);
   delay(iDlit);
   nosound();}
```

Лабораторная работа №2.

Система команд процессора, ее связь с кодами команд

Система команд любого процессора должна удовлетворять таким требованиям как удобство программирования и эффективный доступ к ресурсам. Набор команд определяется организацией самого процессора. Вам предлагается изучить и практически освоить систему команд микропроцессора типа Pentium. Оцените ее: какие Вы видите достоинства и недостатки.

Работа включает шесть этапов: изучение кодирования команд с использованием экранного отладчика TD, знакомство со специальными (особенными) командами процессора, работающими с адресами; выполнение строковых команд при работе с логическими адресами; передача данных из одного места физической памяти в другое и контрольный вопрос на перекодирование команд.

Основное внимание уделено тому, чтобы студент осознал связь мнемоники команды с кодом, который лежит в память, и с действием команды, которое может вызвать изменения указателя команд, флагов, регистров, указателя стека и самого стека.

Цель работы: 1. Изучение системы команд процессора Pentium.

2. Изучение способов кодирования команд.

Информация: Система команд. Формат двухадресной команды.

Используйте один из учебников по Ассемблеру и шаблоны программ.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ:

1. Связь мнемоники с кодом и действием команды
2. Все изменения, которые происходят при выполнении команды
   - указатель команд (переход ?)
   - вычисление адресов
   - вычисление результата
   - флаги
   - указатель стека и сам стек - (... может быть что-нибудь еще ??)
3. Как кодируется команда (КОП, постбайт, адресная часть, префиксы)

ЧТО НУЖНО СДЕЛАТЬ, чтобы убедиться, что Ваше "осознание" пунктов 1-3 совпадает с действительностью:

1. С помощью СИ + TD, а также любой книги с системой команд запишите и выполните команды одной из колонок и поймите результат.

		1	2	3	4	5
- арифметическую операцию	add	inc	sub	cmp	mul	
- логическую операцию		or	not	xor	and	test
- команду передачи управления/ветвления	je		jb	jl	loop	jg
- команду сдвига	shl	sar	rol	ror	rcl	

Для каждой команды:

а) Запишите команду в СИ, используя встроенный ассемблер. Оттранслируйте. Перейдите в TD.

б) Определите двоичный код, выделите поля команды и заполните таблицу. Используя разные способы обращения к операнду, представьте все поля команды.

Таблица двоичных кодов команд

Мнемоника	Префикс	КОП	Постбайт	Смещение	Непоср.операнд
Add ax,bx					
Add ax,[bx]					
Add bx,ax					
Add ax,[bx+2]					
Rep Add bx,ax					
or					
je					
shl					

2. Выполните операции пересылки массива в массив с использованием команд: LOOP, LEA.

3. Напишите программу пересылки данных из одного массива в другой с одной из строковых команд: LODS, MOVS, STOS с использованием префикса повторения REP и подготовки нужных регистров.

4. Напишите программу пересылки данных из одного массива в другой в модели памяти LARGE. Обязательно используйте команду LDS (LES). Укажите, как используются регистры.

5. Напишите программу с использованием строковых команд, которая выполняет действия по переносу содержимого массива, начинающегося с физического адреса B8000 в массив - с адреса B9000. Покажите в

отладчике как изменяются регистры процессора при передаче данных, и как заполняется память.

6*. Разработайте блок-схему и напишите программу поиска символа в видеопамяти на первых 4-х видеостраницах.

7. Ответьте преподавателю на индивидуальные вопросы вида:

- Какой код имеет команда: ADD EAX, [EBX+4*ECX+4].
- Какая команда закодирована кодом: 0202h.
- Какие числа получаются при выполнении команд SAR, SHR, ROR над числом A00Ah.

Система команд

Для каждого типа ЭВМ своя система команд. Так для PC команды описаны в данном пособии и литературе [1-4]. Каждый из изучающих компьютер должен иметь соответствующую книгу по системе команд. Однако имеется много общего в командах для разных компьютеров, и это общее будет рассмотрено ниже.

Количество команд для разных типов ЭВМ колеблется от малых десятков до сотен. В таком множестве разобраться достаточно трудно, поэтому для рассмотрения команд их разбивают на группы (классифицируют). В разных книгах классификация сделана по-разному. Выделяют от 3 до 10 групп команд. Проведем двухуровневую классификацию.

1. Команды пересылки.

1.1. Пересылки общего назначения MOV,L**,LD*,LOD* (от Load - загрузить), ST* (от Store – сохранить). Передают слово/байт данных из одной части ЭВМ в другую без изменения. Иногда в эту группу включают также и команды ввода-вывода для ЭВМ, у которых область адресов внешних устройств включена в общее адресное пространство.

1.2. Пересылки из/в стек: PUSH (втолкнуть),POP (вынуть).

1.3. Пересылки двоичных слов, представляющих собой адреса операндов или части (компоненты) адресов.

1.4. Пересылки между элементами вычислительного ядра (регистры процессора, элементы памяти) и периферийными устройствами.

2. Команды обработки.

2.1.Арифметические.

Минимальный набор арифметических команд очень мал. Это например, сложение – ADD, инвертирование – COM/NOT, прибавление 1 – INC. Все остальное можно сделать, комбинируя эти команды. Однако в современном микропроцессоре арифметических команд обычно больше:

- SUB, NEG, ASR, ASL, DEC, SAR, SAL

- ADC,SBC/SBB операции с переносом C (carry-bit) для выполнения действий с повышенной точностью, когда операнд занимает несколько слов;

- SXT / SBW,SWD расширение знака.

- Команды расширенной арифметики (расширенной для 16 бит проц.): MUL, DIV (обратите внимание, со знаком или без)

- FADD,FSUB,... команды плавающей запятой.

2.2.Логические (это команды побитовой обработки)

- BIS / OR - поразрядное логическое сложение

- BIC / AND - поразрядное логическое умножение

- XOR / XOR - поразрядное исключающее или (ЛИБО)

2.3.Сдвиги

ROR,ROL / ROR,ROL,RCR,RCL - циклические сдвиги. При циклическом сдвиге то, что выходит за границу разрядной сетки, помещается в освобождающуюся позицию на другом конце операнда.

ASR,ASL / SAR,SAL - арифметические сдвиги. Эта разновидность сдвига осуществляется таким образом, что результат оказывается эквивалентен умножению (при сдвиге влево) или делению (при сдвиге вправо) операнда на основание системы счисления, т.е. на 2.

SHL,SHR - логические сдвиги. При выполнении логических сдвигов биты, «выдвигаемые» из разрядной сетки, теряются, а противоположный конец операнда заполняется «нулями».

3. Проверки и передача управления.

Эти команды позволяют реализовать конструкции:

IF ... THEN ... ELSE; REPEAT ... UNTIL;

WHILE ... DO; FOR ... DO; GOTO ...

Для выполнения этих функций в каждом процессоре есть регистр состояния, разряды которого устанавливаются или сбрасываются в зависимости от свойств результата предыдущей операции. Анализируются:

- равенство нулю

- знак !

- перенос !

- арифметическое переполнение ! - эти биты называют флагами (смотри регистр флагов)

3.1. Команды проверки

- TST / - проверка операнда

- BIT / TEST - проверка отдельных битов (логическим умножением)

- CMP / CMP - сравнение операндов

Кроме того, признаки устанавливаются после выполнения многих команд. Обратите внимание на то, что нет единых правил поведения признаков. (В DEC - машинах при пересылке признаки изменяются, а в IBM PC - нет).

3.2. Команды ветвления по условию.

- B** / J*** от слов (branch / jump). Их может быть 10...30 штук, например: BNE - переход, если не равно 0.

3.3. Команда безусловной передачи управления

- JMP / JMP Адресация делается такой, чтобы можно было "прыгнуть" в любое место программной памяти ("длинная" адресация).

3.4. Команда организации цикла

Позволяет организовать в программе структуры

REPEAT ... UNTIL; WHILE ... DO; FOR ... DO более простым способом, чем с помощью команд ветвления.

- SOB / LOOP*

3.5. Команда обращения к подпрограмме (вызов процедуры).

- JSR(CALL) / CALL

4. Команды ввода - вывода.

В системе команд процессора i8086 есть отдельные команды I/O

- IN - команда ввода из ВНУ
- OUT - команда вывода на ВНУ

5. Системные команды (команды управления процессором).

- разрешение/запрещение прерываний
- работа с регистром состояний (с флагами)
- работа с расширенной памятью
- взаимодействие с другими процессорами в многопроцессорной системе.

Формат двухоперандной команды

[Префикс] КОП [постбайт адресации] [смещение] [непоср.операнд]

Префикс Длина 1 байт. (Всего 5 префиксов для X86)

1.	Переназначения сегмента	**add es:[bx],ax**	
2.	Повторения действия для строковых команд	**REP, REPZ, REPNZ, REPNE**	rep movsb
3.	Блокировки	**Lock**	lock rep movsb
4.	Размера адреса		
5.	Размера операнда		

КОП - код операции. Длина 1 байт. 0-й бит КОП во многих (но не во всех) командах показывает, производится ли операция со словом (=1) или с байтом (=0). 1-й бит КОП в двухадресных командах указывает, какой из операндов является приемником.

Постбайт адресации. Длина 1 байт. Постбайт адресации показывает, где находятся операнды. Один из операндов (первый) может быть расположен в регистре (регистровая адресация) или в произвольной ячейке ОЗУ (все способы адресации кроме непосредственной). Второй операнд может

находиться в теле команды (непосредственная адресация) или в регистре (регистровая адресация). Каждый из операндов может быть как источником так и приемником (за исключением непосредственной адресации: непосредственный операнд может быть только источником). Структура системы адресации несимметрична.

```
    7   6     5   4   3     2   1   0
!   mod   !     reg     !     r/m     !
!-----!------!----!----!----!----!-----!----!
```

Поля **mod** и **r/m** задают место расположения первого операнда. Поле **reg** задает положение второго операнда.

Значения поля mod:

11 - операнд в регистре (при остальных mod операнд в ОЗУ, а регистры, на которые указывают поля mod и r/m, содержат компоненты адреса операнда)

10 - смещение два байта (без знака)

01 - смещение один байт (со знаком)

00 - смещение в команде отстутствует

Смещение. Длина 1 байт (при mod-01) или 2 байта(при mod=10).

Непосредственный операнд. Длина 1 или 2 байта

Кодирование регистров полями reg и r/m при mod=11(т.е. при регистровой адресации)

Табл.1

reg или r/m	Байт	Слово
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Кодирование способа вычисления адреса при адресации в память с использованием полей mod и r/m)

Табл.2

r/m	mod=00	mod=01 или 10
000	BX+SI	BX+SI+смещение
001	BX+DI	BX+DI+смещение
010	BP+SI	BP+SI+смещение
011	BP+DI	BP+DI+смещение
100	SI	SI+смещение
101	DI	DI+смещение
110	direct	BP+смещение
111	BX	BX+смещение

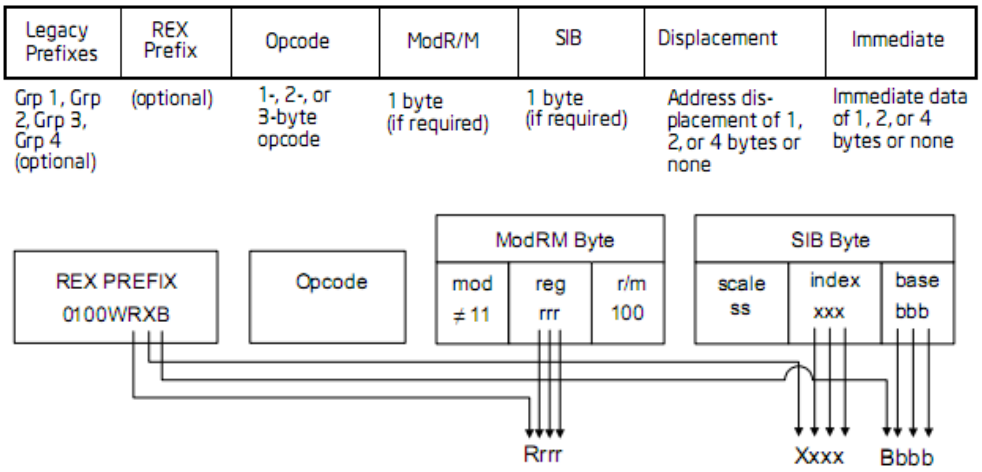
Смещение. Длина 1 байт (при mod-01) или 2 байта(при mod=10).
Непосредственный операнд. Длина 1 или 2 байта
Таким образом, длина команды лежит в пределах от 1 до 10 байт.

В 32-х разрядных процессорах для кодирования расширенных регистров появился в формате после постбайта **SIB байт**. Байт SIB включает в себя следующие поля:

- Поле ss, которое занимает 2 старших бита байта, определяет масштабный коэффициент.

- Поле index, которое занимает следующие 3 бита за полем ss, задает для индексного регистра номер регистра.
- Поле base, которое занимает младшие три бита байта, задает номер базового регистра.

В 64-х разрядных процессорах для кодирования номеров 64-х разрядных регистров появился в формате REX- байт.



Формат 64-х разрядных команд.

Более подробную информацию можно получит в [5].

Пример программы строковой передачи

```
int A[10]={0,1,2,3,4,5,6,7,8,9};
int B[10]={1,1,0,0,0,0,0,0,0,0};
asm {
cld  //сброс флага направления
lea si,A
lea di,B     };
//Распечатайте значения индексных регистров до передачи
printf ("\n  is, di = %x  %x ", _SI, _DI);
asm mov cx,1   ;
asm rep movsb  ;
//Распечатайте значения индексных регистров после передачи
printf ("\n is, di = %x  %x ", _SI, _DI);
printf ("\n Вывод массива после строковой передачи");
for (int i=0; i<10; i++)
{ printf (" \n  %d ", B[i]);    }
   getch();
```

Лабораторная работа №3.

Ассемблер и функции BIOS

Язык Ассемблера специфичен для каждого типа процессора, так как включает в себя совокупность символических обозначений процессорных команд и способов адресации. Несмотря на специфичность, в языках Ассемблера для разных процессоров достаточно много общего, как в форме (в синтаксисе) так и в содержании отображаемых конструкциями языка понятий, поскольку и в различных процессорах также имеется много одинаковых, либо похожих свойств.

Изучение программирования на языке Ассемблера включает в себя три компоненты.

1) Собственно синтаксис Ассемблера, мнемоники команд процессора и способов адресации.

2) Управление процессом трансляции (директивы Ассемблера), позволяющее программисту получить программу с нужными свойствами, например, задать требуемое расположение частей программы в памяти и т.п.

3) Изучение набора и свойств сервисов используемой операционной системы (стандартных подпрограмм ОС, доступных прикладному программисту).

Нам в данном курсе будет нужен только п.1) Ассемблерные мнемоники и фрагменты программ призваны показать, как можно использовать возможности, предоставляемые аппаратурой процессора.

Цель работы: 1. Знакомство с организацией программ на Ассемблере и со средой разработки AsmTool.
2. Изучение основных прерываний BIOS.

Информация: Структура программного модуля. Пример простой программы DOS вывода текстовой строки. Использование системных прерываний.

.

ЗАДАНИЕ НА РАБОТУ

1. Загрузите и запустите шаблон простой программы (смотри в приложении) в среде разработки AsmTool. Убедитесь в успешной работе.

2. Напишите программу на Ассемблере пересылки одноразрядных данных из одного массива в другой с использованием строковых команд. Выведите данные на экране компьютера с использованием 21 прерывания BIOS функция 2.

```
A DB 0,1,2,3,4,5,6,7,8,9
B DB 10 DUP(0)
cld
lea si,A
lea di,B
asm mov cx,10   ;
asm rep movsb  ;
```

3. Модифицируйте программу таким образом, чтобы передавались и выводились 1-, 2-, трехразрядные данные. Для преобразования двоичного кода в ASCII используйте команду деления на 10.

4*. Напишите программу на Ассемблере ввода и вывода на экран многоразрядных чисел.

5*. Напишите программу на Ассемблере ввода, заполнения массива и вывода чисел в формате с плавающей запятой.

Структура программного модуля.

Для языка ассемблера предложения, составляющие программу, могут представлять собой синтаксические конструкции четырех типов [1, 2].

- Команды (инструкции) представляют собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд процессора.
- Макрокоманды — это оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями.
- Директивы являются указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении.
- Комментарии содержат любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

Директивы - команды, не переводящиеся непосредственно в машинные инструкции, а управляющие работой компилятора. Набор и синтаксис их значительно разнятся и зависят не от аппаратной платформы, а от используемого компилятора (порождая диалекты языков в пределах одного семейства архитектур). В качестве "набора" директив можно выделить:

определение данных (констант и переменных);

управление организацией программы в памяти и параметрами выходного файла;

задание режима работы компилятора;

всевозможные абстракции (т.е. элементы языков высокого уровня) - от оформления процедур и функций до условных конструкций и циклов;

макросы.

Команды на языке ассемблера имеет следующий вид:

метка префикс команда(КОП) операнды ; комментарий

- Имя метки — символьный идентификатор. Значением данного идентификатора является адрес первого байта предложения программы, которому он предшествует.
- Префикс — символическое обозначение элемента машинной команды, предназначенного для изменения стандартного действия следующей за ним команды ассемблера.
- Код операции (КОП) — это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора.
- Операнды — части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия.

Пример команды на языке ассемблера

```
L1:  mov  r1, #12A9h  ;Загрузка регистра константой
```

Пояснение: Данный оператор содержит команду пересылки, которая загружает константу 12A9h в регистр процессора r1. Константа задана программистом в виде шестнадцатиричного числа.

Общая структура программы.

```
.386p ;тип процессора
.model flat, stdcall ; модель памяти и вызова подпрограмм
;объявление включаемых (заголовочных) файлов, макросов,
макроопределений,
; также внешних определений
.data
; Инициализированные данные
.data?
; неинициализированные данные
.const
; константы
.code
; исполняемый код
End <метка точки входа>
```

Модель памяти в Windows всегда flat и никакой другой не может быть. Вызов подпрограмм обычно всегда stdcall, стандарт вызова почти всех API функций. Секция .data секция с инициализированными данными она включается в исполняемый файл. Секция .data? секция с

неинициализированными данными, она не включается в исполняемый файл и появляется только тогда, когда программа загружается в память. Секция .const секция констант. Секция .code содержит исполняемый код программы. В конце программы всегда должно стоять слово end, которая задаёт точку входа программы, т.е. место с которого начнётся выполняться программа. Секции .data, .data? имеют полный доступ. Разумеется, секции .const и .code имеют атрибут доступа - только чтение. Секция .const наиболее редко встречается в программах, так как константы можно задавать с помощью макроопределений.

Пример простой программы DOS вывода текстовой строки.

Основные два формата исполнимых файлов в DOS — COM и EXE. 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 килобайта. Файлы типа EXE содержат заголовок, в котором описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти, и т.д. EXE-файл может иметь любой размер. Формат EXE также используется для исполнимых файлов в различных версиях DOS-расширителей и Windows, но со значительными изменениями.

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

```
   DOSSEG
   .MODEL TINY
   .STACK 100h
   .DATA
Message  DB 13,10,'Hi Привет! ',13,10,'$'
   .CODE
   mov    ax,@Data
   mov    ds,ax            ; установить регистр DS таким
                           ; образом, чтобы он указывал
                           ; на сегмент данных
   mov    ah,9             ; функция DOS вывода строки
   mov    dx,OFFSET Message       ; ссылка на сообщение
"Привет!"
   int    21h              ; вывести "Привет!" на экран
   mov    ah,4ch           ; функция DOS завершения
                           ; программы
   int    21h              ; завершить программу
   END
```

Псевдокоманда — это директива ассемблера, которая приводит к включению данных или кода в программу, хотя сама она никакой команде процессора не соответствует. Некоторые псевдокоманды:

DB — определить байт;
DW — определить слово (2 байта);
DD — определить двойное слово (4 байта);
DF — определить 6 байт (адрес в формате 16-битный селектор: 32-битное смещение);
DQ — определить учетверенное слово (8 байт);
DT — определить 10 байт (80-битные типы данных, используемые FPU).

Если нужно заполнить участок памяти повторяющимися данными, используется специальный оператор DUP, имеющий формат
счетчик DUP (значение).
Например, вот такое определение:

```
table_512w     dw   512 dup(?)
```

создает массив из 512 неинициализированных слов, на первое из которых указывает переменная table_512w. В качестве аргумента в операторе DUP могут выступать несколько значений, разделенных запятыми, и даже дополнительные вложенные операторы DUP.

Использование системных прерываний.

Программы на языке Ассемблера, реализующие операции ввода/вывода, используют функции системных прерываний. Команда системного прерывания INT 21H передает управление в DOS для выполнения широкого набора функций, в том числе и связанных с операциями ввода/вывода. Наиболее часто используемые функции прерывания INT 21H представлены в Таблице 1. Команда системного прерывания INT 10H осуществляет передачу управления в BIOS для реализации операций ввода с клавиатуры и вывода на экран. Наиболее часто используемые функции прерывания INT 10H представлены в Таблице 2.

Таблица 1. Функции прерывания INT 21H

Код в AH	Функция	Входные регистры	Выходные рег.
1	Ожидание ввода символа и отображение его на экране		AL - символ
2	Изображение символа на экране	DL - символ	
5	Печать символа	DL - символ	
9	Изображение строки на экране	DS:DX –адрес строки	
A	Чтение строки с клавиатуры	DS:DX –адрес буфера для хранения строки	
2A	Чтение системной даты		CX –год, DH – месяц, DL-день
2B	Установка системной даты	CX – год, DH –месяц, DL - день	AL=0,если дата верна, AL=FF, если ошибочна
2C	Чтение системного времени		CH-часы, CL-мин, DH-сек, DL-сотые доли сек
2D	Установка системного времени	CX-часы, мин, DX- секунды	AL=0 –верно, AL =FF-ошибочно
4C	Возврат в диспетчер DOS	AL – 00 (код возврата)	

ПРИМЕЧАНИЯ:

Функция 9: Строка символов должна заканчиваться $.

Функция 2B: При установке даты можно использовать год - до 2099, месяц- 1...12, день- 1...31.

Функция 2D: При установке времени используются значения: часы – 0...23, мин-0...59, сек- 0...59, сотые доли сек- 0...99.

Таблица 2.Функции прерывания INT 10H.

Код в AH	Функция	Входные регистры	Выходные рег.
0	Задание режима экрана	AL - код режима	
2	Перемещение курсора в Y, X	DH - Y (строка 0...24), DL - X (столбец 0...79), BH - номер страницы(0...4)	
3	Чтение положения курсора	BH - номер страницы(0...4)	DH,DL - Y,X
6	Прокрутка активной страницы вверх	AL - число строк , CH,CL - координаты верхнего левого угла, DH,DL - координаты нижнего правого угла, BH – атрибут строки пробелов	
7	Прокрутка активной страницы вниз	То же , что и для функции 6	
8	Чтение символа в текущей позиции курсора	BH - номер страницы(0...4)	AL- символ, AH- его атрибут
9	Запись символа и нового атрибута в текущую позицию	AL- символ BH - номер страницы(0...4) BL –атрибут символа CX –счетчик символов	
A	Запись символа без изменения атрибута в текущую позицию курсора	То же, но без BL	
E	Вывод символа на экран и перемещение курсора в следующую позицию	То же, что и для 9	

ПРИМЕЧАНИЯ:

Функция 0: AL:= 2 –текстовый ЧБ, 80x25; AL:=3 –текстовый ЦВ, 80x25;

Функция 6: Содержимое AL определяет число остающихся строк, строки внизу заполняются пробелами; при AL=0 происходит очистка всего экрана.

Функции 6 и далее предполагают использование байта-атрибута изображаемых символов и фона.

Примеры использования системных прерываний.

ПРИМЕР 1. Чтение системной даты.

```
MOV  AH, 2AH  ; функция чтения даты.
INT    21H
```

Для вывода даты на экран необходимо извлечь и преобразовать в ASCII код значения: год – из регистра CX, месяц – из регистра DH, число – из регистра DL.

ПРИМЕР 2. Вывод строки символов.

Предположим, что преобразованная в коды ASCII дата хранится в оперативной памяти в переменной DAT, определенной в сегменте данных:

```
DAT DB ‘              ’, 0DH, 0AH, ‘$’
MOV AH, 09H ; функция изображения строки на экране.
LEA DX, DAT ; загрузка адреса строки символов.
INT 21H
```

Следует отметить, что символ $ обязателен и является ограничителем области вывода, а коды 0DH и 0AH обеспечивают перевод строки и возврат каретки печатающего устройства, если следующее сообщение должно начинаться с новой строки. Символом $ в строке символов можно разделить фрагменты, которые должны выводиться отдельно. Например, из строки:

```
MONTH  DB  ‘январь $ февраль $ март $’
```

При загрузке в DX адреса буквы ‘ф’ будет выведено только слово ‘февраль’.

ПРИМЕР 3. Ввод символа с клавиатуры и изображение его на экране.

```
MOV AH, 01H ;функция ввода символа с клавиатуры.
INT 21H
```

ПРИМЕР 4. Устанавливаем новый обработчик прерывания на 8 вектор.

AH = 25h
AL = номер прерывания
DS:DX = вектор прерывания: адрес программы обработки прерывания
INT 21H

```
push    ds
        mov     ax,cs
        mov     ds,ax
        lea     dx,[kernel]   // Адрес нового обработчика
        mov     ah,025h
        mov     al,008h
        int     021h
        pop     ds
```

Без прерывания MSDOS напрямую изменяем значения таблицы векторов прерываний:

```
call    _di                 // Блокируем прерывания
       push    ds
       xor    bx,bx
       mov    ds,bx
       lea    bx,[cs:kernel]    // Адрес нового обработчика
       mov    [ds:0020h],bx
       mov    [ds:0022h],cs
       pop    ds
       call    _ei                 // Разрешаем прерывания
```

Лабораторная работа №4.

Способы адресации и сегментная организация памяти

Понятие «Способ адресации» включает:

1. Способ кодирования адреса в адресном поле команды
2. Условное обозначение (синтаксис) способа адресации при записи команды на языке ассемблера
3. Алгоритм вычисления исполнительного адреса по информации, содержащейся в адресном поле, а также в других элементах процессора, имеющих отношение к вычислению адреса (хранящих адресную информацию, компоненты адреса)

Выбор какого-либо способа определяется задачей минимизации длины команды, доступом к операнду и удобством программирования. В процессоре типа Pentium используются следующие способы адресации:

1. Непосредственная адресация add ax,12
2. Абсолютная адресация. call proc
3. Регистровая адресация. add ax,bx
4. Косвенно-регистровая адресация. add ax,[bx]
5. Базовая адресация (адресация по базе)add ax,[bx+disp]
6. Базово-индексная add ax,[bx+si]
7. Базово-индексная со смещением add ax,[bx+si+disp]
8. Относительная адресация. je next-line

Цель работы: 1. Изучение особенностей способов адресации процессора.

2. Связь способов адресации и времени выполнения команды.
3. Влияние моделей памяти на результат выполнения программы.

Информация: Способы адресации. Возможности измерения интервалов времени в Win.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ:

1. Как кодируется способ адресации
2. Какова схема вычисления адреса
3. Какие узлы процессора (адресные регистры и т.п.) содержат компоненты адреса
4. Как влияет выбор данного способа адресации на время выполнения команды.

5. Как эмулировать команду со сложным способом адресации последовательностью более простых.

6. Как расположены сегменты памяти при трансляции программы в СИ. Какие модели памяти используются. Как работать с разными сегментами и как пересылать данные из одного сегмента в другой.

7. Как используются префиксы переназначения сегментов

ЧТО НУЖНО СДЕЛАТЬ:

1. На примере одной из команд реализуйте все способы адресации (регистровая, косв.-регистр., индексная/базовая, базово-индексная, базово-индексная со смещением)

2. На примере одной команды измерьте время выполнения команды при разных способах адресации (add ax,bx; add ax,[bx+si+5]; add ax,[bx]; add [bx],ax). Определите тактовую частоту процессора, частоту работы ОЗУ, влияние КЭШ. Учитывайте время пустого цикла.

3. Напишите программу заполнения четных строк двумерного массива, находящегося в сегменте данных, числами с использованием двух вариантов адресации. Программа должна показывать преимущества многокомпонентных способов адресации или строковых команд (stos).

4. Оттранслируйте пример (п.3), написанный на СИ, в двух моделях памяти - поймите что изменилось, а если ничего не изменилось, то почему?

5. Напишите программу на Асс, результат работы которой меняется в зависимости от используемой модели памяти. Например, чтение данных из сегмента кода, сделайте так, чтобы читались одинаковые или разные данные.

Способы адресации

Для организации адресации в процессоре используются четыре 16-разрядных сегментных регистра, которые указывают на начала четырех сегментов памяти. Местоположение операнда в сегменте определяется смещением (offset), которое вычисляется при трансляции адреса в зависимости от способа адресации. Местоположение сегментов в ОЗУ определяется в программе. Если Вы работаете в среде СИ, то положение сегментов задается при выборе модели памяти. Один и тот же операнд в зависимости от его описания и предыстории может находиться в любом из сегментов.

1. Непосредственная адресация.

Если операнд является заранее известной константой, то его значение удобно помещать прямо в команде за кодом операции. Никаких вычислений для определения адреса операнда делать не надо. Операнд может быть разной длины (занимать разное число ячеек/байтов).

2. Абсолютная (прямая) адресация.

В команде за КОП помещается заранее известный адрес операнда. Обратите внимание, что абсолютная адресация является косвенным вариантом для непосредственной адресации.

3. Регистровая адресация.

Операнд находится в одном из регистров процессора. Обращение к регистру процессора гораздо быстрее, чем к ячейке ОЗУ. Указатель регистра занимает более короткое поле чем адрес.

4. Косвенно-регистровая адресация.

Адрес операнда находится в одном из регистров процессора. Тот факт, что адрес операнда находится в регистре, позволяет вычислять или модифицировать этот адрес, при этом один и тот же участок программы может обрабатывать разные элементы данных (находящиеся в разных адресах).

Многокомпонентные способы адресации

Во всех предыдущих случаях адрес операнда хранился где-то в одном месте и выбирался сразу весь, целиком.

Для того, чтобы:

а) можно было вычислять адрес в ОЗУ более изощренно/гибко;

б) можно было сократить разрядность кода адреса в команде или в регистре (для этого сокращенного кода также используют термин "логический адрес") физический адрес может формироваться путем объединения нескольких частей (компонент), хранимых в разных местах.

5. Базовая адресация (адресация по базе)

Ее идея состоит в том, что физический адрес получается путем суммирования адреса, сформированного любым из ранее рассмотренных однокомпонентных способов адресации и содержимого одного из регистров процессора, называемого регистром базы.

6. Базово-индексная bp di

7. Базово-индексная со смещением bx + si + disp

Эти способы адресации используются при обращении к двумерным структурам данных и позволяют уменьшить количество действий при вычислении адреса.

8. Относительная адресация.

В качестве одной из компонент при формировании адреса используется счетчик команд. Применяется в командах условного и безусловного перехода, в командах цикла и вызова подпрограмм.

Возможности измерения интервалов времени в Win

1. Команда **RDTSC** (Read Time Stamp Counter)

Команда возвращает количество тактов, прошедших с момента подачи напряжения или сброса процессора.

Команда RDTSC состоит из двух байтов: $0F 31. Она возвращает в регистры EDX:EAX 64-битное значение time-stamp счетчика. Счетчик хранится в 64 битном регистре MSR.

Выполнение инструкции RDTSC доступно с любого уровня привилегий, при условии, что в регистре CR4 флажок "Time stamp disable" установлен на "0". В противном случае выполнение данной инструкции доступно только с нулевого уровня привилегий.

2. Функция API SetTimer().

Пример программы заполнения двумерного массива

```
void main(void)
{
int A[90]={0,1,2,3,4,5,6,7,8,9};
// Заполняем  массив на уровне Ассемблера
asm mov si,0
e2:
asm {
    lea bx,A
    mov  cx,10
    mov  ax,0
    };
e1: asm mov [bx],ax // Выберите способ адресации
    asm add bx,2
    asm loop e1
    asm add si,10  // Задается 5 элементов в строке
    asm mov ax,100
    asm cmp ax,si
    asm ja e2
// Вывод двумерного массива
 for (i=0; i<20; i=i+4)
    {   printf (" \n   ");
     for (int j=0; j<4; j++)
      {   printf (" ");
      }
    }
}
```

Лабораторная работа №5.

Подпрограммы и передача параметров.

Модульность программирования на аппаратном уровне поддерживается организацией механизма подпрограмм. Этот механизм включает следующие элементы: команды обращения к ПП, передачу параметров, запоминание состояние процессора в стеке, программное прерывание и др. Аппаратные средства должны поддерживать вложенность ПП, быстрое обращение к ПП, обращение к ПП, расположенных в любых участках памяти ЭВМ. В работе предлагается ряд заданий, которые позволяют изучить и практически освоить работу с подпрограммами.

Цель работы: 1. Изучение особенностей команд работы с подпрограммами процессора Pentium.

3. Эмуляция команд работы с расширенными регистрами.

2. Освоение методов передачи параметров в подпрограмму. Анализ использования стека.

Информация: Связь с ПП по управлению. Передача параметров в ПП. Организация стека.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ (дать ответы на вопросы себе и преподавателю):

1. Как организована связь с ПП по управлению:
   - команды call, ret;
   - внутрисегментная передача, межсегментная передача, косвенная и прямая адресация.
2. Как используется стек для сохранения адреса возврата и состояния процессора. Структура стекового кадра.
3. Передача параметров (связь по данным) с использованием:
   - регистров
   - стека
   - общей области памяти (как еще ?).

ЧТО НУЖНО СДЕЛАТЬ:

1. Оформите одно из заданий, приведенных ниже, в виде подпрограммы с параметрами на встроенном ассемблере в среде СИ.

Операнды находятся в памяти (отведите для них место, используя средства языка Си).

Ваш фрагмент должен не только выполнять требуемое действие (формировать результат операции), но и обеспечивать верные значения флагов (такие же, как при выполнении соответствующей "короткой" команды из системы команд процессора). Вы можете "отследить" не все флаги, достаточно будет, если Вы обеспечите верные значения для двух флагов. Возьмите обязательно ZF и какой-нибудь второй по собственному выбору (но только такой, который ведет себя при выполнении команды нетривиально).

*Разрядность операндов задайте как параметр. Смотрите в качестве прототипа шаблон программы эмуляции арифметического сдвига вправо.

2. Реализуйте обращение к ПП с передачей параметров через стек:
- внутрисегментное (модель памяти Tiny),
- межсегментное (модель памяти Large).

Приведите таблицу заполнения стека в обоих случаях. Отметьте отличия.

3. Реализуйте передачу параметров в ПП и результата из ПП с использованием:
- регистров,
- общей области памяти.

Задания по теме ПОДПРОГРАММЫ:

1. Напишите фрагмент программы на встроенном ассемблере - сдвига арифметического вправо (влево).
2. Напишите фрагмент программы на встроенном ассемблере. Сдвиг циклический вправо (влево).
3. Напишите фрагмент программы на встроенном ассемблере. Сдвиг логический вправо (влево).
4. Напишите фрагмент программы на ассемблере - инкремент двойного слова. Сравните Ваш способ с тем, как это делает компилятор.
5. Напишите фрагмент программы на ассемблере - декремент двойного слова. Посмотрите, как это делает компилятор.
6. Напишите фрагмент программы на ассемблере - смена знака двойного слова.
7. Напишите фрагмент программы на ассемблере - сложение двух двойных слов.

8. Напишите фрагмент программы на ассемблере - сравнение двух двойных слов.
Для шустрых:
9. Напишите фрагмент программы на ассемблере - умножение двух двойных слов.

Связь с ПП по управлению

Обращение к подпрограмме и выход из нее выполняется двумя командами: CALL, RET. При обращении / возврате надо обеспечить:
а) передачу управления в любое место памяти, поэтому "длинная" адресация;
б) возврат в то место, откуда был вызов (т.е. место вызова должно автоматически запоминаться)
в) запоминание промежуточных результатов, имеющихся к моменту вызова (содержимое регистров процессора, а при рекурсивном вызове процедуры надо запоминать и промежуточные результаты работы самой процедуры).

Для запоминания всего этого используется стек (участок ОЗУ или специальное ОЗУ со стековой адресацией). Часть вышеперечисленной информации запоминается автоматически при выполнении команды CALL, а сохранение оставшегося - дело программиста.

Внутрисегментная команда CALL (NEAR) (модель памяти Tiny) осуществляет передачу управления только внутри текущего 64-килобайтного командного сегмента. В стеке сохраняется содержимое только регистра IP.

Межсегментная команда CALL (FAR) (модель памяти Large) может передавать управление любой процедуре в пределах мегабайтового адресного пространства микропроцессора. В стеке сохраняется содержимое регистров CS и IP.

Если процедура имеет (определена) атрибут NEAR, то команда CALL помещает смещение адреса следующей команды в стек. Если процедура имеет атрибут FAR, то команда CALL помещает в стек содержимое регистра CS, а затем смещение адреса.

После сохранения адреса возврата команда CALL загружает смещение адреса метки вызываемой процедуры в указатель команд IP. Если процедура имеет атрибут FAR, то загружается также номер блока метки вызываемой процедуры в регистр CS.

Команда RET заставляет микропроцессор возвратиться из процедуры в программу, вызывающую эту процедуру. RET извлекает адрес возврата из стека. Если процедура имеет атрибут NEAR, то команда RET извлекает из стека одно слово и загружает его в указатель команд IP. Если процедура имеет атрибут FAR, то команда RET извлекает из стека два слова: сначала

смещение адреса для загрузки в указатель команд IP, а затем номер блока для загрузки в регистр CS.

Передача параметров в ПП. Организация стека.

Передачу параметров в ПП и результата из ПП можно осуществить, через:
регистры; стек; общую область памяти.

Стек удобен для временного хранения данных (содержимого регистров и ячеек памяти). Вершина стека - ячейка в сегменте стека, адрес которой содержится в указателе стека SP. Т.к. стек "растет" по направлению к младшим адресам памяти, то первое помещаемое в стек слово запоминается в ячейке стека с наибольшим адресом, следующее на 2 байта ниже и т.д. Регистр SP всегда указывает на слово, помещенное в стек последним. Например, если ассемблерная ПП вызывается из СИ:
RES=AAA(a1,a2) // a1.a2 - имеют тип long
То стек будет выглядеть следующим образом:

для функции с атрибутом FAR	для функции с атрибутом NEAR	комментарий
sp+0A	sp+8	a2 старшее слово
sp+8	sp+6	a2 младшее слово 1
sp+6	sp+4	a1 старшее слово 1
sp+4	sp+2	a1 младшее слово 1
sp+2	нет	регистр CS вызывающей программы
sp	sp	регистр IP вызывающей программы

Для а) сохранения-восстановления контекста, б) связи по данным (передачи параметров и возврата значений) в) выделения памяти под локальные переменные используется фрагмент стека, называемый ***стековым кадром***.
Пример на Си:

```
int Calc (int x, int y) {
    int iX, iY;
    iX=2*x+y;
    iY=3*y-x;
    return iX*iY; }
```

Эта процедура принимает два параметра **x** и **y** и возвращает результат вычисления несложного арифметического выражения в глобальную переменную **z**. В процедуре объявлены две локальные переменные. Далее

приведен результат компиляции этой процедуры после его дизассемблирования полноэкранным отладчиком TurboDebugger.

AB1_3.CALC: begin	
cs:0000 55 push bp	Сохранение контекста
cs:0001 89E5 mov bp,sp	В bp базовый адрес стекового кадра
cs:0003 83EC04 sub sp,0004	Выделение места под локальные переменные
AB1_3.15: iX:=2*x+y;	
cs:0006 8B460A mov ax,[bp+06]	Доступ к первому параметру
cs:0009 D1E0 shl ax,1	
cs:000B 034608 add ax,[bp+08]	Доступ ко второму параметру
cs:000E 8946FE mov [bp-02],ax	Доступ к локальной переменной
AB1_3.16: iY:=3*y-x;	
cs:0011 8B4608 mov ax,[bp+08]	Доступ ко второму параметру
cs:0014 8BF0 mov si,ax	
cs:0016 D1E0 shl ax,1	
cs:0018 01F0 add ax,si	
cs:001A 2B460A sub ax,[bp+06]	Доступ к первому параметру
cs:001D 8946FC mov [bp-04],ax	Доступ к локальной переменной
AB1_3.17: z:=iX*iY;	
cs:0020 8B46FE mov ax,[bp-02]	Доступ к локальной переменной
cs:0023 F766FC imul word ptr [bp-04]	Результат оставляем в AX
cs:0026 C47E04 les di,[bp+0A]	Берет третий параметр – адрес результата
cs:0029 268905 mov es:[di],ax	Запись результата процедуры
AB1_3.18: end;	
cs:002C 89EC mov sp,bp	Освобождение локальных переменных
cs:002E 5D pop bp	Восстановление контекста
cs:002F C20800 ret 0008	Возврат из процедуры с освобождением места

Поддержка возможности рекурсивного вызова

Рекурсивный вызов (когда ПП вызывает сама себя) иногда бывает полезен. Классический пример - вычисление факториала с использованием соотношения

N ! = N * (N-1) !

Другой пример - решение задачи о "Ханойской башне".

Рекурсия при вызове подпрограммы может быть как прямая (ПП вызывает сама себя) так и цепная (ПП-а А вызывает ПП-у В, та в свою очередь вызывает ПП-у С, а затем ПП-а С вызывает ПП-у А).

Проблем при рекурсивном вызове несколько:

1) Запоминание адресов возврата - каждый экз. ПП имеет свой адрес возврата

2) Каждый экземпляр ПП должен иметь собственные наборы внутренних переменных

3) Иногда требуется, чтобы экземпляры ПП, вызванные позднее, имели доступ к внутренним переменным ранее вызванных экземпляров.

Решение всех этих проблем обеспечивается, когда при вызове ПП в стеке выделяется блок ячеек (стековый кадр) для параметров и временных переменных очередного экземпляра ПП (как в примере, рассмотренном выше). Адресацию внутри стекового кадра удобно делать относительно какого-либо его адреса. Для этого перед вызовом ПП параметры помещаются в стек::

```
push    par1
push    par2
call    subr
```

в начале ПП применяется (как мы уже увидели) типовая последовательность команд:

```
subr: push  bp    ; запоминание старого значения регистра базы bp
      mov  bp,sp  ; передача в bp адреса начала стекового кадра
```

Теперь можно к локальным переменным ПП обращаться, используя адресацию относительно регистра базы:

```
mov  ax,[bp+4]
mov  bx,[bp+6]
```

Для обеспечения возможности использовать параметры и переменные "внешних" экземпляров ПП при рекурсивном вызове, стековый кадр должен содержать ссылки на "предыдущие" стековые кадры.

В системе команд процессора могут быть команды, облегчающие формирование стекового кадра. Например, в Pentium есть команды ENTER и LEAVE.

Пример шаблона программы 1.

```
/*================================================*\
Эмуляция сложных (несуществующих) команд
   Студенты гр 000.0 Иванофф, Петрофф, & Сидорофф
Задание: эмуляция команды арифметического сдвига вправо
операнда с повышенной разрядностью. Операнд с повышенной
разрядностью находится в массиве iMas1, количество слов в
длинном операнде содержится в переменной iNWords, число
разрядов, на которое надо сдвинуть длинный операнд, содержится
в переменной iNShift.
\* ================================================ */
//  Глобальные переменные
int iNWords=3;    // Количество слов в операнде
int iNShift=4;    // На сколько битов сдвинуть
int iMas1[10]={0x1234, 0x2000,0x1000,0x800,0x400,0x200};
```

```
            // Массив для  операнда
void main(void)
{//Используем регистр SI для перебора слов длинного операнда
//  Заносим начальный адрес операнда в адресный регистр
     asm mov bx, iNShift //  Заносим счетчик сдвигов
//Далее пойдет цикл, на каждой итерации которого
//выполняется сдвиг длинного операнда на один бит
 cykl_po_sdwigam:
// ------------------------------------------
asm{
  mov cx, iNWords//  Заносим счетчик слов
  mov si, offset iMas1 //Прибавим к адресу младшего слова
  add si, iNWords      // сдвигаемого операнда удвоенную
  add si, iNWords      // длину операнда
                        //  Получили адрес старшего слова+2
  test word ptr[si-2],0x8000//Проверяем старшее слово (при
                           // этом С-бит всегда очищается
  jns cykl_po_slowam //  Если отрицательно,
  stc               // устанавливаем С-бит для верного
       }            // расширения знака при сдвиге
/* ------------------------------------------- *\
  mov ax, [si]   - Это другой вариант расширения знака в С
  rcl ax
\* ------------------------------------------- */
//  Теперь цикл по словам длинного операнда
  cykl_po_slowam:     //
asm {
  dec si; dec si //  Модифицируем адрес
  rcr word ptr[si],1 //  Сдвигаем циклически
  loop cykl_po_slowam   //  Организуем цикл
      }
  asm dec bx            //  Считаем сдвиги
  asm jg cykl_po_sdwigam
//  Определяем признак "нуля"
  asm mov cx, iNWords   //  Заносим счетчик слов
  asm xor dx, dx        //  Для признака нуля
zero:
asm {
  or dx, [si]            //  Признак "нуля"
  inc si; inc si         //  Модифицируем адрес
  loop zero
        }
  asm test dx,0xFFFF         //  Признак "нуля"
  asm nop}
```

Пример шаблона программы 2.

```
// Тема - ПОДПРПОГРАММЫ. Передача параметров через стек.
#include <stdio.h>
#include<conio.h>
```

```
#include<iostream.h>
/*
При вызове функции в Си, она интерпретируется как команда Call
в Assembler. При этом передаваемые в функцию параметры,
заносятся в СТЕК по принципу справа налево. Затем заносится IP
следующей команды, идущей за Call. (внутрисегментный вызов
процедуры)
Выход из функции осуществляется с помощью команды RET.
*/

//Функция, выполняющая сложение, параметры передаю через стек
void addition(long sl_1,long sl_2, long &sum)
{
/*   asm{
       mov     AX,[BP+4] //AX - младшие разряды первого
слагаемого
       mov     DX,[BP+8] //DX - младшие разряды второго
слагаемого
       mov     SI,[BP+12] //SI - сумма младших разрядов
       add     AX,DX      //сложение младших разрядов
       mov     [SI],AX    //результат косвенно в SI
       mov     AX,[BP+6]  //AX - старшие разряды первого
слагаемого
       mov     DX,[BP+10] //DX - младшие разряды второго
слагаемого
            //сложение старших разрядов
       mov     [SI+2],AX   //результат в SI
    }*/
}
void main ( void )
{long a,b;
long sum;
cout<<"\n Введите первое слагаемое:  ";
cin>>a;
cout<<" Введите второе слагаемое:  ";
cin>>b;
addition(a,b,sum);
cout<<"a + b =  "<<sum;
getch();
}
```

Пример шаблона программы 3.

```
// Тема - ПОДПРПОГРАММЫ. Передача параметров через стек.
// встроенный ассемблер Visual Studio

#include <stdio.h>
#include <iostream>
using namespace std;
long long var1, var2;
```

```
void shift_stack(long long a, long long &b)
{
	_asm
	{
		mov eax, [ebp + 8]
		shl eax, 1
		mov ebx, [ebp + 12]
		rcl ebx, 1
		mov esi, [ebp + 16]
		mov[esi], eax
		mov [esi+4], ebx
	}
}
int main()
{
	setlocale(LC_ALL, "rus");
	long long a;
	long long b;
	cout << "Введите значение: ";
	cin >> a;

	//Передача через стек
	shift_stack(a, b);
	cout << "Сдвиг через стэк: " << b << endl;

	//Передача через регистры

	//Передача через глобальные переменные

	return 0;
}
```

Лабораторная работа №6.

Подпрограммы, программные прерывания и особые случаи.

Действия при прерывании могут быть использованы и для перехода к ПП, предусмотренного программистом (программные прерывания). Для этого в систему команд вводится команда программного прерывания int. Можно вызвать ПП и через организацию особого случая. Прерывания и особые случаи подобны в том, что они заставляют процессор временно приостановить выполнение текущей программы для перехода к ПП. Прикладные программисты обычно не касаются обработки особых случаев, их обрабатывает операционная система. Однако некоторые типы особых случаев имеют отношение к прикладному программированию, и многие операционные системы предоставляют возможность прикладным программам их обрабатывать.

Предлагается реализовать вызов ПП через программное прерывание и особый случай, проанализировать заполнение стека.

Цель работы: 1. Изучение особенностей команд программного прерывания.

2. Освоение методов вызова ПП через программное прерывание и особый случай. Анализ использования стека.

Информация: Типы программных прерываний. Вызов прерывания.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

1. Программное прерывание как средство вызова ПП, достоинства и недостатки.

Команда int - возбуждает программное прерывание, позволяющее передать управление программе обработки прерывания;

Команда iret - возврат управления из программы обработки прерывания

2. Вектор прерывания. Разрешенные и запрещенные вектора.

3. Какие регистры запоминаются в стеке.

4. Отличие от аппаратного прерывания

(В аппаратном прерывании:

- разрешение / запрет прерывания в регистре флагов и во внешнем устройстве;

- необходимость запоминания в стековом кадре состояния всех используемых регистров;

- возможность программирования вложенности прерываний)

5. Отличия особых случаев от других источников прерываний.

6. Каковы источники особых случаев.

ЧТО НУЖНО СДЕЛАТЬ

1. Реализуйте выполнение вашей ПП через программное прерывание (команда int) для чего: запомните старый вектор прерывания, установите новый, введите параметры ПП, выполните прогр. прерывание, обработайте данные в ПП (в программе обработке прерывания), выйдите в основную программу, выведите результат.

2. Проанализируйте заполнение стека.

3. Реализуйте выполнение вашей ПП путем создания особого случая (деление на нуль), соответствующему одному из первых десяти векторов.

4. Проанализируйте заполнение стека. При выходе из обработчика прерываний пропустите команду, вызывающую особый случай .

Типы программных прерываний.

Подобно вызову процедуры прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, находящейся в некоторой ячейке памяти. Но при вызове процедуры микропроцессор исполняет процедуру, а при прерывании программу обработки прерывания.

В то время как вызываемая процедура может иметь атрибуты NEAR и FAR, а ее вызов будет прямым или косвенным, прерывание всегда вызывает косвенный переход к своей программе обработки за счет получения ее адреса из вектора прерывания. Кроме того, вызовы процедуры сохраняют в стеке только адрес, а прерывание еще и флаги.
У микропроцессора есть 3 команды прерывания INT, INTO, IRET.

При исполнении команды INT микропроцессор производит следующие действия:
1. Помещает в стек регистр флагов.
2. Обнуляет флаг трассировки TF и флаг включения-выключения прерываний IF для исключения пошагового режима исполнения команд и блокировки других маскируемых прерываний.
3. Помещает в стек значение регистра CS.
4. Вычисляет адрес вектора прерывания, умножая тип прерывания на 4 (четыре).
5. Загружает второе слово вектора прерывания в регистр CS.

6. Помещает в стек значение указателя команд IP.
7. Загружает в указатель команд IP первое слово вектора прерывания.

После исполнения команды INT в стеке окажутся значения регистра флагов и регистров CS и IP (на макушке стека), флаги TF и IF будут равны 0, а пара регистров CS:IP будут указывать на начальный адрес программы обработки прерывания. Затем микропроцессор начнет исполнение этой программы.

Команда IRET по отношению к прерыванию выполняет ту же роль, что и команда RET для вызова процедуры. Она "откатывает" всю работу исходной операции и заставляет микропроцессор аккуратно выполнить возврат к основной программе. Команда IRET извлекает из стека 3 16-битовых значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр флагов соответственно.

Вызов прерывания.

Для генерации внутреннего прерывания программы в СИ используется встроенная функция Geninterrupt(). Чтобы использовать программное прерывания как средство вызова ПП надо:

1. Сохранить старое значение вектора прерывания.
Old-f0=getvect(0xF0)
2. Установить новую функцию обработки прерывания для вектора прерывания. setvect(0xF0,add)
3. Вызвать процедуру генерации программного прерывания.
geninterrupt(0xF0)
4. Восстановить старое значение вектора. setvect(0xF0,Old-f0)

Пример программы с командой программного прерывания

```
#include<stdio.h>
#include<dos.h>
#include<iostream.h>
#include <conio.h>
void interrupt (*old)(...);      // здесь будем сохранять
старый вектор
void interrupt cmp_int(...)      // а это наш обработчик
{cout<<"Прерывание  ";
getch();}
void main(void)
{ int aa;
  aa=1;
  old=getvect(0xf0);
  disable();
  setvect(0xf0,cmp_int);
  enable();
geninterrupt(0xf0);
  puts("v1=v2");
  setvect(0xf0,old);
  return;}
```

Лабораторная работа №7.
FPU. Кодирование чисел с плавающей запятой. Особые численные значения. Особые случаи.

Типовой формат представления числа с плавающей запятой:

Знак	Порядок	Мантисса

Знак относится ко всему числу. Поле мантиссы содержит ее значение в прямом (не в дополнительном) коде с опущенным неявным битом. Поле порядка содержит сумму истинного порядка и положительной константы, называемой смещением.

Цель работы: 1. Изучение особенностей команд чисел с плавающей запятой.
2. Освоение методов вызова ПП через особый случай чисел с плавающей запятой.

Информация: Прерывания и особые случаи. Численные особые случаи при обработке команд FPU. Кодирование вещественных чисел с одинарной и двойной точностью. Формат слова управления FPU. Слово состояния FPU. Команды управления.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

1. Формат чисел с плавающей запятой.
2. Регистровая модель и команды работы с регистрами FPU
3. Как работает FPU при обработке особых случаев.

ЧТО НУЖНО СДЕЛАТЬ

1. Напишите программу на Acc (см пример): Введите в диалоге три числа, загрузите их в регистры FPU, просуммируйте, выведите результат. Проследите работу FPU в отладчике. Как используются регистры FPU ? Как сказывается ограниченность разрядной сетки мантиссы.

2. Получите в FPU нечисло (NaN), покажите его в регистре.

3. Прочитайте и проанализируйте состояния регистров FPU: регистра состояния и регистра управления.

4. Напишите программу, выполняющую команды FPU, вызывающую и реагирующую на особый случай FPU (обработчик прерывания сообщает о наступлении особого случая по результату анализа регистра состояния).

5. При сдаче работы покажите преподавателю механизм перевода числа, например 16.02, в формат с плавающей запятой.

Фрагмент программы работы с FPU как со стеком.

// Операции с плавающей запятой на уровне Ассемблера

```
asm {   finit
    fld ad
    fld bb
    fadd
    fstp bb
}
cout<<"   Acc  "<<bb;
```

Стандарт на числа ПТ ANSI/IEEE 754-1985

Форматы: одинарная, двойная, расширенная точность

Русское наименование	Короткое вещественное (КВ)	Длинное вещественное (ДВ)	Временное вещественное (ВВ)
Английское наименование	Short Real (SR)	Long Real (LR)	Temporary Real (TR)
Всего битов	32	64	80
Поле порядка	8	11	15
Смещение порядка	27 - 1 = 127	210 - 1 = 1023	214 - 1 = 16383
Диапазон десятичных порядков	-37 ÷ +38	-307 ÷ +308	-4931 ÷ +4932
Поле мантиссы	23	52	64
Точных знач. десятичных цифр	6	15	19
Неявный бит	Опущен	Опущен	Изображается

Численные особые случаи при обработке команд FPU

FPU распознает следующие шесть классов условий численных особых случаев при выполнении численных команд:

I - недействительная операция: ошибка стека, недействительная операция для стандарта IEEE;

Z - деление на нуль;

D - денормализованный операнд;

O - численное переполнение;

U - численное антипереполнение;

P - неточный результат.

При возникновении численных особых случаев процессор выполняет одно из следующих действий:

- FPU может сам обрабатывать особый случай, формируя наиболее приемлемый результат и разрешая численной программе продолжать выполнение;
- осуществляется вызов программного обработчика особого программного случая.

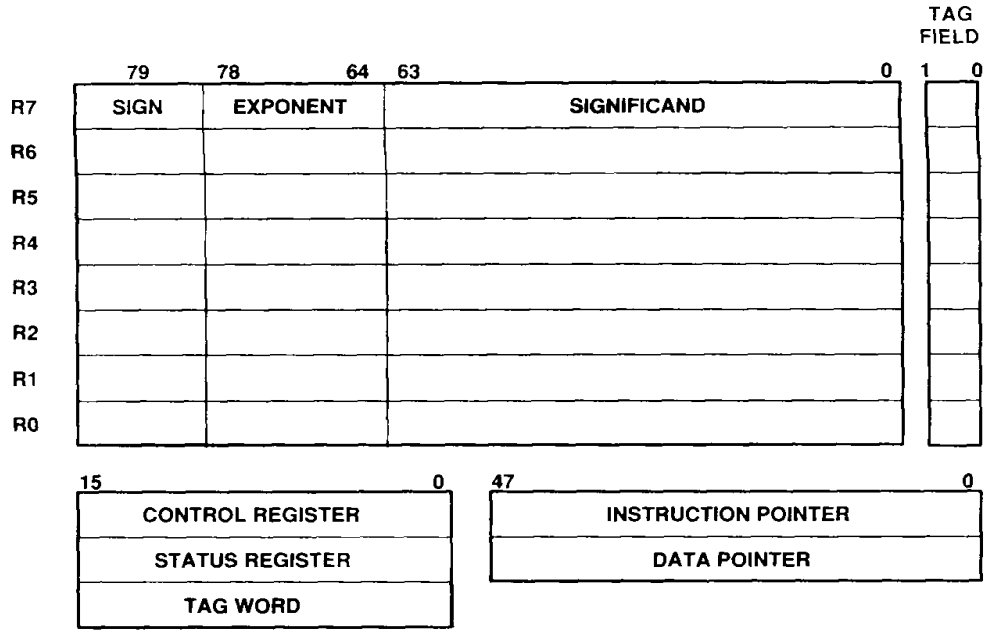
Перечислим особые случаи в порядке убывания приоритета.

Особый случай	Ответ FPU по умолчанию
1) недействительная операция	QNaN (особ. случ. стека) SNaN (недейств операция)
2) деление на нуль	Бесконечность
3) денормализованный операнд	Денормализ. операнд (правила действия – стремится к нормализации)
4)численное переполнение	+Беск., -Беск., Nmax, Nmin В зависимости от режимов округления
5) численное антипереполнение	Денормализ. операнд или 0.
6) неточный результат	Результат округления

Каждое из условий шести особых случаев, указанных выше, имеет соответствующий бит в слове состояния FPU и бит маски в слове управления. Если особый случай маскируется, то процессор по умолчанию выполняет соответствующее действие и продолжает работу. Если особый случай замаскирован (0 в маске регистра управления), то обработчик особого случая немедленно вызывается перед выполнением следующей команды WAIT или неуправляющей команды FPU. В зависимости от значения бита NE в регистре управления CR0 процессора обработчик исключения вызывается или через вектор прерывания 16(NE=1), или посредством внешнего прерывания 0x75 (NE=0).

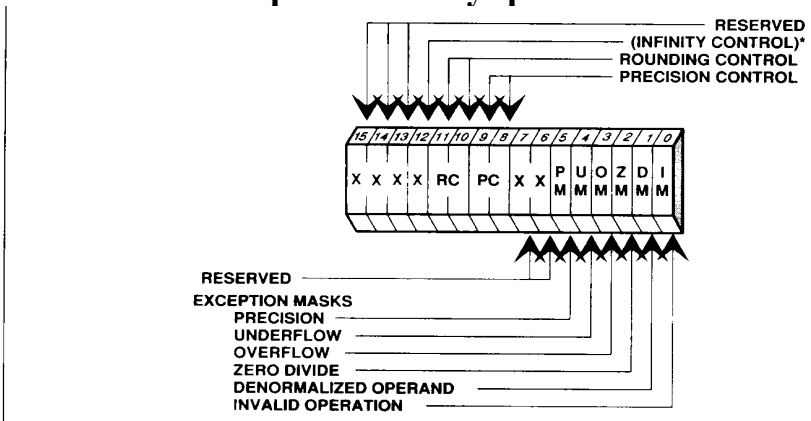
Когда особые случаи замаскированы, FPU может обнаружить множество особых случаев в одной команде, так как продолжает выполнение команды после завершения действий на маскируемый случай.

Архитектура FPU



Процессор с плавающей точкой FPU включает следующие регистры:
- восемь 80-битных регистров, организованных как регистровый стек;
- три 16-битных регистра, содержащих слова управления FPU, слово состояния FPU, слово тэгов;
- указатели ошибок.

Формат слова управления FPU



ROUNDING CONTROL	PRECISION CONTROL
00 —ROUND TO NEAREST OR EVEN	00—24 BITS (SINGLE PRECISION)
01 —ROUND DOWN (TOWARD -∞)	01—(RESERVED)
10—ROUND UP (TOWARD +∞)	10—53 BITS (DOUBLE PRECISION)
11—CHOP (TRUNCATE TOWARD ZERO)	11—64 BITS (EXTENDED PRECISION)

FPU позволяет использовать несколько режимов обработки чисел с плавающей точкой (см. табл), которые выбираются при загрузке слова управления.

Команда **FSTCW** запоминает слово управления. Команда **FLDCW** загружает слово управления.

Значения битов слова управления:

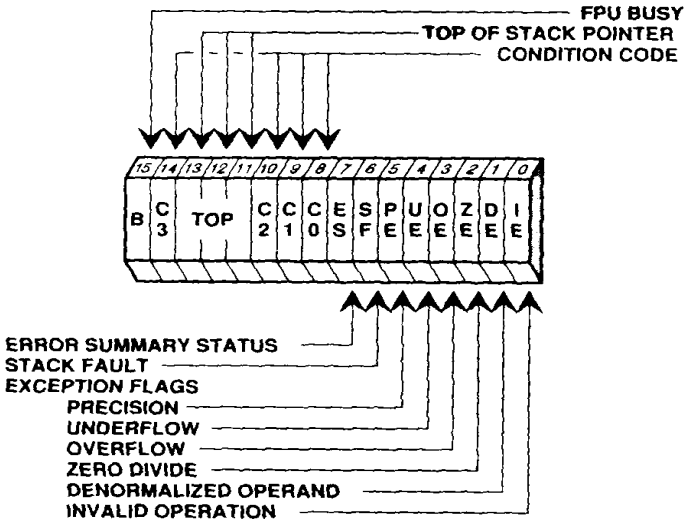
15,14,13 - Зарезервированные
12 - Управление бесконечностью*
11,10 - Управление округлением
9,8 - Управление точностью
7,6 - Зарезервированные

МАСКИ ИСКЛЮЧЕНИЙ

5 - Точность *
4 - Антипереполнение
3 - Переполнение -
2 - Деление на нуль
1 - Денормализованный операнд
0 - Недопустимая операция

Управление округлением	Управление точностью
00 - округление до ближайшего или до четного 01 - округление вниз (к минус бесконечности) 10 - округление вверх (к плюс бесконечности) 11 - усечение (отбрасывание разрядов)	00 - 24 бита (обычная точность) 01 - (зарезервировано) 10 - 53 бита (двойная точность) 11 - 64 бита (расширенная точность)

Формат слова состояния FPU



Слово состояния FPU отражает полное состояние устройства с плавающей точкой. Команда FSTSW AX запоминает слово состояния непосредственно в регистре AX, разрешая процессору просматривать эти коды условия.

Значения битов слова состояния:

15 - FPU занято

13,12,11 - Вершина указателя стека

14,10,9,8 - Код условия

7 - Суммарное состояние ошибок Бит ES(7) устанавливается, если любой немаскированный бит особого случая установлен, иначе он очищается.

6 - Ошибка стека -*

ФЛАГИ ОСОБЫХ СЛУЧАЕВ

5 - Точность ---

4 - Отрицательное переполнение

3 - Переполнение ---

2 - Деление на нуль -*

1 - Денормализованный операнд

0 - Недействительная операция

Значения TOP (биты 13,12,11)таковы:

000 - регистр 0 - вершина стека

001 - регистр 1 - вершина стека

.....

111 - регистр 7 - вершина стека

Четыре бита кода условия FPU (C3 - C0) подобны флагам Центрального процессора: процессор модифицирует эти биты, чтобы отразить результат арифметических операций. Эти биты кода условия используются для условного ветвления.

Команды управления

Команды управления FPU показаны в табл. Они обеспечивают управление FPU на системном уровне. Эти действия включают инициализацию FPU, обработку численных особых случаев и переключение задач. Команды, которые инициализируют FPU, сбрасывают особые случаи или запоминают всю или частично операционную среду FPU, можно представить в следующих двух модификациях:

-ожидать (WAIT) - мнемонике предшествует только один символ F. Эта форма команды проверяет немаскируемые численные особые случаи;

-не ожидать (NO-WAIT) - этой мнемонике предшествует два символа FN. Данная форма игнорирует немаскируемые численные особые случаи.

Команды управления

Мнемоника	Операция
FINIT/FNINIT	Инициализация FPU
FLDCW	Загрузить слово управления
FSTCW /FNSTCW	Запомнить слово управления
FSTSW/FNSTSW	Запомнить слово состояния
FSTSW AX/FNSTSW AX	Запомнить слово состояния в регистре AX
FCLEX/FNCLEX	Сбросить особые случаи
FSTENV/FNSTENV	Запомнить среду
FLDENV	Загрузить среду
FSAVE/FNSAVE	Запомнить полное состояние
FRSTOR	Восстановить полное состояние
FINCSTP	Инкремент указателя стека
FDECSTR	Декремент указателя стека
FFREE	Освободить регистр
FNOP	Нет операции
FWAIT	Ожидание

Для процессора Pentium целочисленное устройство и устройство с плавающей точкой работают параллельно, возможно, что при возникновении особого случая для операций с плавающей точкой процессор разрушит информацию, необходимую для обнаружения особого случая прежде, чем обработчик подобных исключений будет вызван. Использование команды WAIT или FWAIT в требуемом месте программы может предотвратить такую ситуацию.

Кодирование вещественных чисел с одинарной и двойной точностью

Класс	Знак	Смещенный порядок	Мантисса
Положительные нечисла			
QNaN	0	11..11	11..11
	O	11..11	10..00
SNaN	O	11..11	01..11
	0	11..11	00..01
Бесконечность	0	11..11	00..00
Положительные вещественные			
Нормализованные	0	11..10	11..11
	0	00..01	OO..OO
Денормализованные	O	00..00	11..11
	0	00..00	00..01
Отрицательные вещественные			
Денормализованные	1	00..00	00..01
	1	00..00	11..11
Нормализованные	1	00..01	00..00
Бесконечность	0	11..11	00..00
Отрицательные нечисла			
SNaN	1	11..11	01..11
	1	11..11	00..01
QNaN	1	11..11	11..11
	1	11..11	10..00

Одинарный:	8битое -	23 бита-
Двойной:	11битов-	52 бита-

Лабораторная работа №8.

Отладчик

При отладке программ желательно иметь следующие возможности:
1) выполнять отлаживаемую программу до заранее заданного места (задавать точки останова);
2) выполнять отдельные участки по шагам (по одной команде);
3) наблюдать текущее состояние отлаживаемой программы (все ее переменные, включая и вспомогательные, которые не фигурируют явно в программе);
4) иметь возможность принудительно завершить "зависшую" программу.

В качестве инструмента отладки используют специальные вспомогательные программы-отладчики, которые взаимодействуют с отлаживаемой программой.

Цель работы: 1. Изучение особенностей команд, связанных с отладкой.
2. Освоение методов перехода на подпрограмму пошаговой отладки с использованием команды выхода из обработчика прерывания.

Информация: Задание точек останова (breakpoints). Поддержка пошаговой отладки.

ЗАДАНИЕ НА РАБОТУ

ЧТО НУЖНО ОСОЗНАТЬ

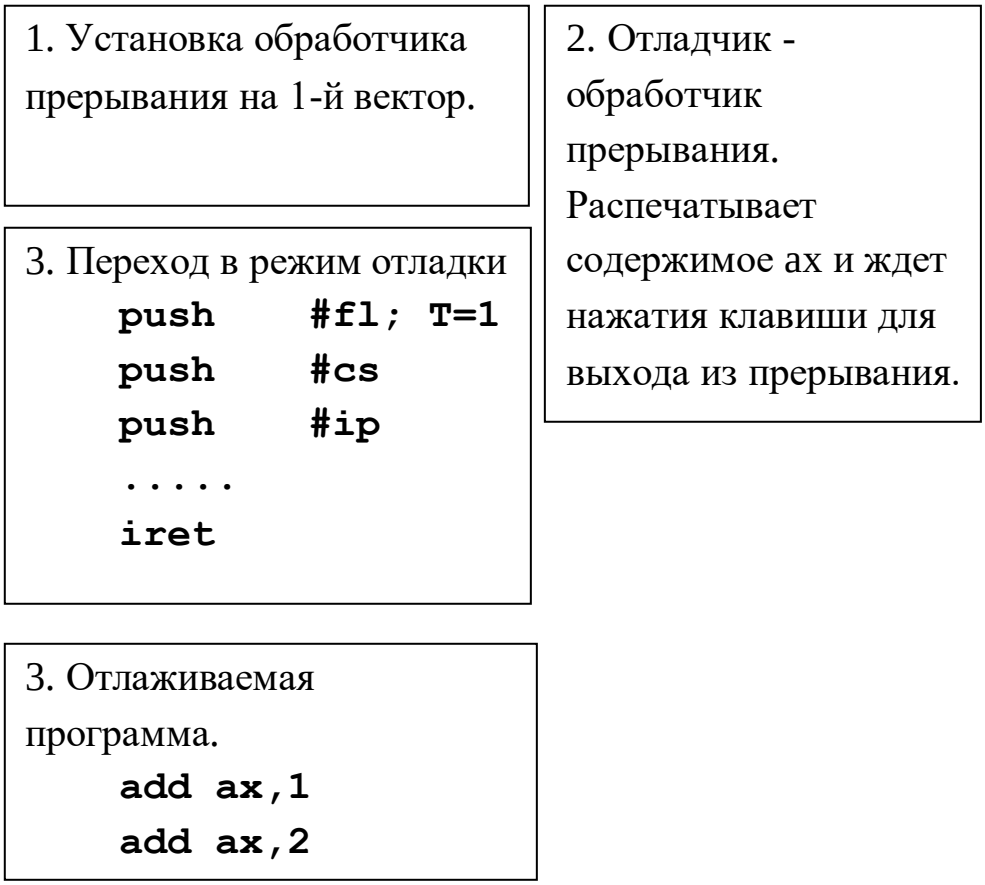
1. Механизм пошаговой отладки, Т-разряд.
2. Построение отладчика.
3. Переход в режим отладки.

ЧТО НУЖНО СДЕЛАТЬ

1. Напишите программу в Borland C или на Ассемблере, которая содержит два фрагмента, один из которых играет роль пошагового отладчика, а второй имитирует отлаживаемую программу. Используйте Т-разряд для организации взаимодействия между фрагментами. Отладчик должен распечатывать содержимое регистра AX и ждать нажатия клавиши. Если не удается построить программу, то смотрим п.3.

2. Докажите преподавателю, что протекает пошаговая отладка.

3. Структурная схема отладчика может быть следующей:



4. Включите в отлаживаемую программу команду int3 - задание точки останова. Определите ее код с использованием TD. Установите отладчик на 3-й вектор. Покажите в отладчике выход на точку останова.

Задание точек останова (breakpoints).

После заданной команды не выполняется следующая команда, а управление передается отладчику.

команда i-1
команда i ее хотим выполнить последней
команда i+1 вместо нее **(переход на отладчик)**,
команда i+2
команда i+3

В системе команд Pentium для точки останова есть специальная команда int3 длиной в 1 байт. Эта команда вызывает прерывание через вектор 3. Отладчик для установки (нескольких) остановов (breakpoint) использует следующую последовательность действий:

1) запоминает для каждой точки останова затираемый один байт
2) записывает в эти точки однобайтовую команду **int3**
3) записывает в **вектор3** адрес перехода на себя (адрес возврата в отладчик)

4) передает управление отлаживаемой программе

Программа выполняется, а если натолкнется на **int3** - произойдет прерывание, в стек запишется адрес, на 1 больше точки останова (по нему можно идентифицировать, какой останов сработал).

5) отладчик должен вернуть на место все сохраненные байты отлаживаемой программы.

Поддержка пошаговой отладки.

В этом случае точкой останова (breakpoint - переход на отладчик) должна быть каждая команда отлаживаемой программы. Описанная выше техника задания точек останова в этом случае оказывается громоздкой, а иногда просто неприменима.

Пошаговую отладку обычно поддерживают в процессорах на аппаратном уровне его разработчики. Типовой прием для этого - наличие в процессоре особого режима, когда **прерывание происходит после выполнения каждой команды**. Если бы такой режим был включен постоянно, то он был бы неработоспособным. Для включения-выключения этого режима обычно используют один из битов регистра состояния процессора. В процессорах Pentium - это бит 8 регистра состояния (TF Trap Flag). Если TF=1, то пошаговое прерывание разрешено, TF=0 соответствует обычному режиму работы.

Теперь необходимо организовать, чтобы TF был равен 1, когда выполняется команда отлаживаемой программы, и сбрасывался в 0 при переходе на отладчик. После того, как отладчик выполнит все действия по индикации состояния отлаживаемой программы, необходимо осуществить переход на очередную ее команду, одновременно установив TF в 1, чтобы после выполнения этой команды вновь произошло прерывание. Эти требования выполняются автоматически, если использовать следующую технику:

Отладчик перед передачей управления на отлаживаемую программу,

1) записывает в вектор пошагового прерывания (номер 01h, адрес 00004h) адрес перехода в отладчик после выполнения команды отлаживаемой программы;

2) записывает в стек (с соответствующим смещением указателя стека): а) слово состояния процессора с установленным TF-битом, б) адрес отлаживаемой программы (CS:IP - 4 байта), с которого надо начать пошаговую отладку;

2) выполняет команду возврата из прерывания **iret**.

```
.....
push    #psw
push    #cs
push    #ip
```

```
        .....
        iret
l1      .....
```
; сюда можем вернуться по прерыванию после выполнения команды.

По команде **iret** из стека загружается в CS:IP адрес "возврата" в отлаживаемую программу, а в SR - слово состояния с установленным TF-битом. Выполняется команда отлаживаемой программы (одна !), после чего происходит прерывание. Теперь выполняются действия, как при любом прерывании: слово состояния с установленным TF-битом и содержимое CS:IP (адрес следующей команды отлаживаемой программы) сохраняются в стеке, из вектора 01h загружается CS:IP (это адрес перехода в отладчик). Теперь отладчик может показать состояние отлаживаемой программы после сделанного шага.

Пример программы отладчика (Т.М. Садайкин)

```
.model large
code segment
assume cs:code, ds:code, es:code, ss:code
org 100h
start:

 ;Перейти на начало
 jmp beg

 print_symbol:
 ;Распечатать символ
  push ax
  push dx
  mov ah, 02h
  mov dl, dh
  cmp dl, 0
  je t2
  int 21h
  t2:
  pop dx
  int 21h
  pop ax
  ret

 print_number:
  ;Распечатать число, base = 10
  push ax
  push bx
```

```
 push cx
 push dx
 mov ax, dx
 mov bx, 10
 mov cx, 0
 getdigits:
  mov dx, 0
  div bx
  inc cx
  add dx, 30h
  push dx
  cmp ax, 0
  jnz getdigits
 mov ah, 02h
 printdigits:
  pop dx
  int 21h
  loop printdigits
 pop dx
 pop cx
 pop bx
 pop ax
 ret

debug:
 ;Сохранить регистры, запретить прерывания
 cli
 push bp
 mov bp, sp
 push ax
 push bx
 push cx
 push dx
 push si
 ;Вывести ip и ax
 mov dx, '['
 call print_symbol
 mov dx, [bp+2]
 call print_number
 mov dx, ','
 call print_symbol
 mov dx, [bp-2]
 call print_number
 mov dx, ']'
 call print_symbol
```

```
 mov dx, etr
 call print_symbol
 ;Ожидать нажатие клавиши
 xor ax, ax
 int 16h
 ;Восстановить регистры, разрешить прерывания
 pop si
 pop dx
 pop cx
 pop bx
 pop ax
 pop bp
 sti
 iret

;Отлаживаемая программа
thread:
 mov ax, 1
 mov cx, 10
 t1:
  add ax, ax
  loop t1
 ret //???????????????

beg:
 ;Сохранить старый обработчик
 mov ax, 3501h
 int 21h
 mov int1, bx
 mov int1+2, es
 ;Установить новый обработчик
 push cs
 pop ds
 mov dx, offset debug
 mov ax, 2501h
 int 21h
 ;Сохранить парамеры для возврата
 pushf
 push offset exit
 ;Установит TF флаг
 pushf
 pop ax
 or ax, 100h
 push ax
 push cs
```

```
  push offset thread
  iret

 exit:
  popf
  ;Установить старый обработчик
  lea dx, int1
  mov ax, 2501h
  int 21h
  ;Выход после нажатия клавиши
  mov ah, 01h
  int 21h
  int 20h

int1 dw 0h, 0h
etr dw 0Ah, 0Dh

code ends
end start
```

Лабораторная работа №9.

Обмен ЭВМ с клавиатурой

В данной работе предлагается изучить программную модель клавиатуры и экспериментально исследовать возможности, которые предоставляет клавиатура пользователю. Рассматриваются: коды, которые передаются из блока клавиатуры в процессорный блок; вопросы синхронизации во времени между нажатиями клавиш и действиями процессора; организация статусных байтов и буфера клавиатуры в ОЗУ; управление клавиатурой со стороны процессора.

Предлагается запустить программу scancode.com и наблюдать значения скан-кодов, которые соответствуют нажатиям и опусканиям клавиш. Запустить программу keyview.com, моделирующую поведение буфера клавиатуры, и наблюдать перемещение указателей головы и хвоста, а также скан-код клавиши и ASCII-код символа, которые заносятся в буфер. В итоге следует написать и отладить программу, которая выполняет следующую последовательность действий (не используя функции DOS или BIOS): ждет нажатия клавиши, читает скан-код нажатой клавиши, ASCII-код нажатой клавиши, адрес в буфере клавиатуры, с которого расположены эти коды. Программа должна показать, что происходит при нажатии клавиш Shift, Ctrl, Alt .

Цель работы: 1. Экспериментально исследовать возможности, которые предоставляет клавиатура пользователю.

Информация: Структура подключения клавиатуры. Порты для работы с клавиатурой. Описание команд.

ЗАДАНИЕ НА РАБОТУ

1. Вопросы:

Разберитесь, как организована связь клавиатуры с процессором в PC и выясните:

1.1. Что (какие коды) передается из блока клавиатуры в процессорный блок (и куда именно: в какие программно доступные узлы) ?

1.2. Как организована синхронизация во времени между нажатиями клавиши действиями процессора ?

1.2.1. Как процессор "узнает", что нажата очередная клавиша ?

1.2.2. Как контроллер клавиатуры узнает, что процессор считал предыдущий код, и можно посылать следующий ?

1.2.3. Что произойдет, если клавиши нажимаются быстро одна за другой, а процессор по каким-то причинам "не успевает" обрабатывать эти нажатия ?

1.3. Как организован буфер клавиатуры в ОЗУ ?

- Что записывается в буфер клавиатуры по нажатию клавиши ?
- Какая программа это делает ?
- Что происходит при нажатии клавиш Shift, Ctrl, Alt ?
- Что происходит при нажатии клавиш CapsLock, NumLock, ScrollLock ?

Каким образом (и кто?) "узнает" о том, что незадолго до этого была нажата одна из этих клавиш

1.4. Какие есть возможности по управлению клавиатурой со стороны процессора ?

2.Методическое задание:

1. Придумайте способ (методику), позволяющий наблюдать (с помощью экранного отладчика либо с помощью написанной Вами программы) "поведение" буфера клавиатуры при нажатии выбранных Вами клавиш (эти клавиши будем называть экспериментальными). Затруднение состоит в том, что для управления отладчиком тоже приходится манипулировать клавишами (управляющие нажатия).

2. Придумайте, как отличить управляющие нажатия от экспериментальных. Для нескольких экспериментальных нажатий зафиксируйте соответствующие изменения в буфере клавиатуры и проинтерпретируйте их.

3. Практическое задание 1:

3.1. Запустите программу **scancode.com** и и наблюдайте значения скан-кодов, которые соответствуют нажатиям и опусканиям клавиш.

3.2. Запустите программу **keyview.com**, моделирующую поведение буфера клавиатуры и наблюдайте перемещение указателей головы и хвоста, а также скан-код клавиши и ASCII-код символа, которые заносятся в буфер.

3.3. Напишите и отладьте программу, которая выполняет следующую последовательность действий (не используя функции DOS или BIOS):

а) ждет нажатия клавиши (каким способом она может узнать, что нажата клавиша.)

б) по нажатию клавиши читает (откуда можно читать?)

-скан-код нажатой клавиши

-ASCII-код нажатой клавиши

-адрес в буфере клавиатуры, с которого расположены эти коды

в) выводит указанные значения на экран.

4. (Задание для шустрых) Напишите программу, которая выполняет ту же функцию, что и в п.3.3, но не читает буфер клавиатуры, а вызывается по прерыванию от клавиатуры. Программа вначале должна "перехватить" вектор прерывания типа 9, а перед завершением восстановить его. Если это

задание вызовет трудности, Выполните его позже, когда подробнее познакомитесь с контроллером прерываний.

5. Практическое задание 2:

Напишите программу, которая запрещает прерывание от клавиатуры на заданное время (например на 10 с), при этом другие прерывания не должны быть запрещены, например, должны идти системные часы.

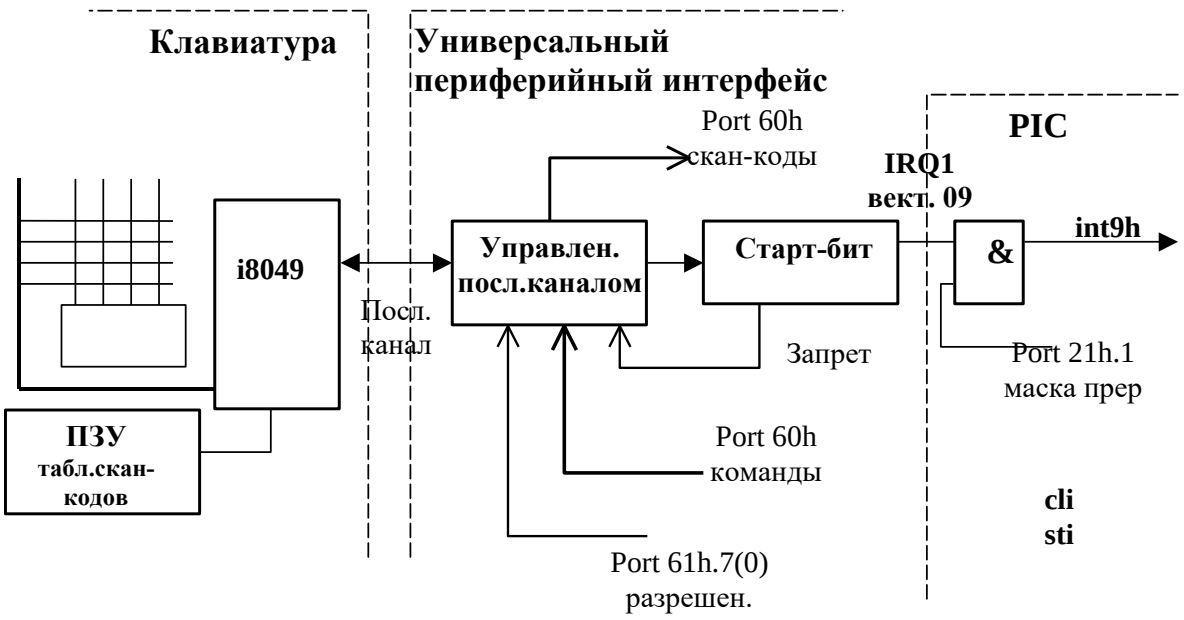
6. Практическое задание 3:

Напишите программу, которая по нажатию выбранных Вами клавиш зажигает/гасит индикаторы CapsLock, NumLock, ScrollLock.

7. Практическое задание 4:

Напишите программу, которая запрашивает с клавиатуры значения задержки и частоты автоповтора, а затем устанавливает соответствующие величины для клавиатуры.

Схема подключения клавиатуры



Все горизонтальные линии матрицы подключены через резисторы к источнику питания +5 В. Клавиатурный компьютер (однокристалльный контроллер 8049) имеет порта - входной и выходной. Входной порт подключен k горизонтальным линиям матрицы, а выходной - вертикальным. Устанавливая но очереди на каждой из вертикальных линий уровень напряжения, соответствующий логическому нулю контроллер опрашивает состояние горизонтальных линий. Если нажатых клавиш нет, уровень напряжений на всех горизонтальных линиях соответствует логической 1.

Если оператор нажмет на какую-либо клавишу, то соответствующие вертикальная и горизонтальная линии окажутся замкнутыми. Когда на этой вертикальной линии процессор установит значение логического нуля, то уровень напряжения на горизонтальной линии также будет соответствовать логическому нулю. Как только на одной из горизонтальных линий появится уровень логического нуля, контроллер фиксирует нажатие на клавишу. Он посылает в центральный компьютер запрос на прерывание и номер клавиши в матрице. Аналогичные действия выполняются, когда оператор отпускает нажатую ранее клавишу.

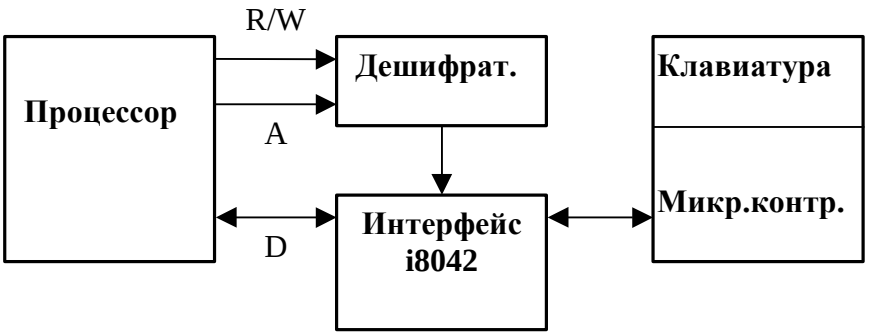
Номер клавиши, посылаемый контроллером, однозначно связан с распайкой клавиатурной матрицы и не зависит от обозначений, нанесенных на поверхность клавиш. Этот номер называется скан-кодом (Scan Code). Слово scan ("сканирование") подчеркивает, что клавиатурный компьютер сканирует клавиатуру для поиска нажатой клавиши. Но программе нужен не порядковый номер нажатой клавиши, а соответствующий обозначению на этой клавише ASCII-код. Этот код не зависит однозначно от скан-кода, т. к, одной и той же клавише могут соответствовать несколько значений ASCII-кода. Это зависит от состояния других клавиш. Например, клавиша с обозначением '1' используется еще и для ввода символа '!' (если она нажата вместе с клавишей SHIFT). Поэтому все преобразования скан-кода в ASCII-код выполняются программным обеспечением. Как правило, эти преобразования выполняют модули BIOS. Для использования символов кириллицы эти модули расширяются клавиатурными драйверами.

Если нажать на клавишу и не отпускать ее, клавиатура перейдет в режим автоповтора. В этом режиме в центральный компьютер автоматически через некоторый период времени, называемый периодом автоповтора, посылается код нажатой клавиши. Режим автоповтора облегчает ввод с клавиатуры большого количества одинаковых символов.

Следует отметить, что клавиатура содержит внутренний 16- байтовый буфер, через который она осуществляет обмен данными с компьютером.

Клавиатура, благодаря наличию встроенного в нее контроллера может программироваться. Вы можете программно установить скорость автоповтора и величину задержки до его начала, а также управлять светодиодами клавиатуры.

Контроллер клавиатуры генерирует запрос прерывания (IRQ 1) при каждом нажатии или отпускании клавиши. Прерывание IRQ 1 вызывает переход по вектору INT 09H и обрабатывается подпрограммой BIOS.



Порты для работы с клавиатурой

Для работы с клавиатурой используются порты с адресами 60h,64h и 61h.

Порт 60h при чтении содержит скан-код последней нажатой клавиши.

Порт 61h управляет не только клавиатурой, но и другими устройствами компьютера, например работой встроенного динамика. Порт доступен как для чтения, так и для записи. Для нас важен самый старший бит этого порта. Если в старший бит порта 61h записать значение 1, клавиатура будет заблокирована, если 0 разблокирована.

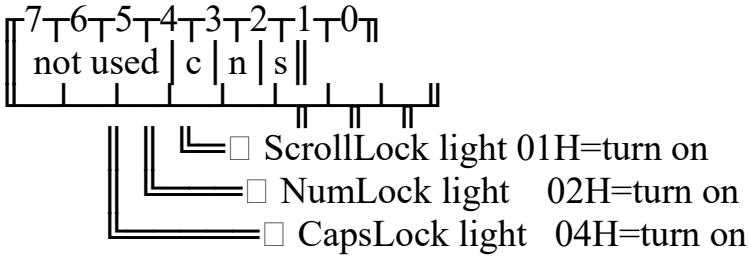
Так как порт 61h управляет не только клавиатурой, при изменении содержимого старшего бита необходимо сохранить состояния остальных битов этого порта. Для этого можно сначала выполнить чтение содержимого порта в регистр, изменить состояние старшего бита, затем выполнить запись нового значения в порт:
in al, 61h or al, 80h out 61h, al

Компьютер позволяет управлять скоростными характеристиками клавиатуры, а также зажигать или гасить сведиоды на лицевой панели клавиатуры - Scroll Lock, Num Lock, lps Lock.

Для команд, требующих посылки двух байтов, таких как установление скорости автоповтора, между двумя командами out следует сделать небольшую задержку

Описание некоторых команд

0FFh Сброс клавиатуры и запуск внутреннего теста
0FEh Повторить последнюю передачу
0FDh-0F7H (NOP)
[0FDh - Вызов прерывания IRQ1]
0F6h Привести клавиатуру в исходное состояние и разрешить сканирование
0F5h Привести клавиатуру в исходное состояние и запретить сканирование
0F4h Сбросить буфер клавиатуры и начать сканирование
0F3h Задать скорость автоповтора и задержку до ее начала
0EDh Управление светодиодами. Второй байт команды имеет формат:



Буфер клавиатуры

Указатели 0:041Ah - голова (последняя записанная)
0:041Ch - хвост (последняя несчитанная)

- Сам буфер находится в ячейках 0:041Eh...0:043Dh
- В буфер запись производит обработчик прерывания 09h.
- Считывание из буфера может осуществлять программа, используя функции DOS либо читая буфер напрямую.
- Как определить, пуст ли буфер? - сравнить указатели
- Как очистить буфер? - приравнять указатели

Буферы статуса клавиатуры: **0:0417h, 0:0418h**

Пример программы работы с клавиатурой.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <bios.h>
#define Port8042 0x60

long far * pTime=(long far*)0x46C;   // Указатель на
счетчик тиков

// Определим указатели начала и конца буфера

int far * pHeadPtr=(int far  *)0x41A;   // Указатель на
указатель головы буф.кл.
int far * pTailPtr=(int far  *)0x41C;   // Указатель на
указатель хвоста буф.кл.
unsigned char far * pBuf;

void main()
{
// Задание N1.
  clrscr();

  cout << "\n Задание N1:\n";
  cout << "\n Нажимайте клавиши для получения кодов!";
```

```
  cout << "\n Пробел - идем дальше.\n";
  char cScan,cAsci;
  int iAdres;
  do
  {
   // Загрузить пару регистров, например ds:si значением
0x40:0x1A
     asm{
          push ds  //
          push si          //  Сохранили регистры
          push di          //
          mov ax,0x40      //
          mov ds,ax        //   Загрузили сегмент
          mov si,0x1A      // и указатели на голову
          mov di,0x1C      // и на хвост
          }
     wait_kbd:
     asm{
        mov ax,[si]      //   Сравниваем указатели -
        cmp ax,[di]      // т.е. ждем нажатия
        je wait_kbd
     // Загрузить регистр di значением 0x1C
     // Сравнить указатели
     // Прочитать указатель
        mov si,[si]
        }
     iAdres=_SI;
     // Прочитать значение из буфера
     asm mov ax,[si]
     // Теперь _AL и _AH содержат скан-код и ASCII-код
     asm pop di
     asm pop si
     asm pop ds
     cScan=_AH;
     cAsci=_AL;
     printf("Адрес = %x   Скан = %d      ASCII = %d
\n",iAdres,cScan,cAsci);
  } while(getch() != 32);
  cout << "\n Конец первого задания. Press any key...\n";
  getch();
       delay(1000);
      asm in al,0x60
      cScan=0; // ??????
      printf(" Скан = %x \n",cScan);
//    goto met;}
```

Лабораторная работа №10.

Мультизадачность

Мультипрограммирование предназначено для повышения пропускной способности вычислительной системы путем более равномерной и полной загрузки всего ее оборудования, в первую очередь процессора. При этом скорость работы самого процессора и номинальная производительность ЭВМ не зависят от мультипрограммирования.

Основные черты мультипрограммного режима:

- в оперативной памяти находятся несколько пользовательских программ в состояниях активности, ожидания или готовности;
- время работы процессора разделяется между программами, находящимися в памяти в состоянии готовности;
- параллельно с работой процессора происходит подготовка и обмен с несколькими устройствами ввода-вывода.

Большинство микропроцессоров, в том числе Pentium, имеют средства аппаратной поддержки мультизадачности. Микроконтроллеры не имеют средств аппаратной поддержки мультизадачности. В этом случае требуется на программном уровне обеспечить переключение задач при наступлении разных событий. В данной работе предлагается реализовать этот подход.

Цель работы: Получить опыт организации простой мультизадачности (без приоритетов).

Информация: Понятие мультизадачности. Многозадачность и операционная система. Контекст. Переключение контекста. Статическое и динамическое распределение ресурсов.

ЗАДАНИЕ НА РАБОТУ

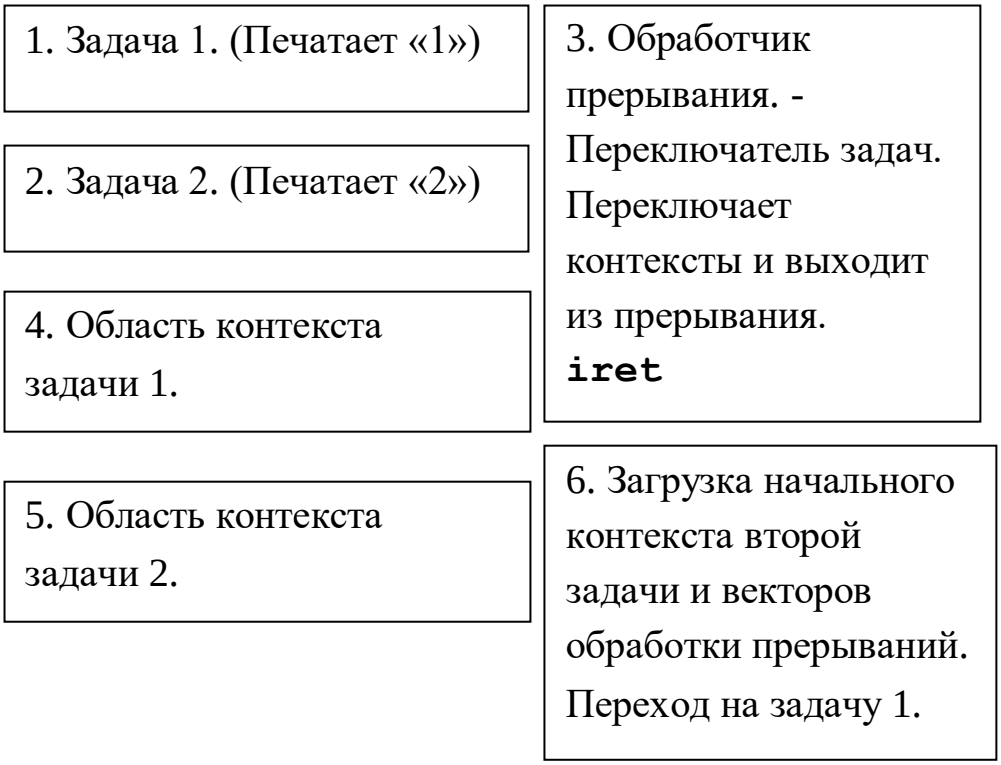
ЧТО НУЖНО ОСОЗНАТЬ

1. Виды контекста задачи (полный, короткий).
2. Методы сохранения и восстановления контекста.
3. Механизм переключения задач.

ЧТО НУЖНО СДЕЛАТЬ

1. Организуйте три задачи. Выделите три области (статические) для сохранения контекста задач. Загрузите начальный контекст. Обеспечьте переключение между задачами по внешнему событию или таймеру, например по нажатию клавиши.

2. При необходимости воспользуйтесь приведенной ниже структурой элементов механизма переключения задач. На рисунке показаны следующие элементы: две задачи, их контексты, обработчик прерывания – диспетчер задач, программа перехода в режим мультизадачности.



3*. Установите статические приоритеты задач. Организуйте переключение с использованием приоритетов.

Понятие мультизадачности (многозадачность)

Понятие «задача» (task) в первом приближении аналогично понятию «программа». Когда говорят о многозадачном режиме, имеют в виду возможность одновременного (параллельного) выполнения нескольких программ. Истинная параллельность возможна лишь в многопроцессорной системе, когда каждая программа выполняется на своем ресурсе.

В однопроцессорной системе несколько программ могут разделять процессорное время. При переключении с одной задачи на другую требуется сохранять контекст прерываемой задачи, чтобы можно было впоследствии продолжить ее выполнение. Это обычно делает операционная система, причем для этого совсем не обязательно наличие какой-либо специальной аппаратной поддержки.

Чем может быть полезна многозадачность.

В большей части случаев достигается повышение производительности труда человека, например в следующих ситуациях:

Выполнение длинных рутинных операций (форматирование текста, долгая передача через модем, копирование длинных файлов и т.п.)

Работа одновременно с несколькими приложениями, когда выходные данные одних служат входными для других. Например, связка FineReader (сканирование и распознавание текста) + PhotoEditor (сканирование и редактирование изображений) + AcrobatReader (чтение документов в формате .pdf) --> MsWord (окончательное оформление документа, содержащего выдержки из документации, из сканированного текста и рисунки).

Использование процессорного времени фоновой задачей в том случае, когда текущая задача ожидает предоставления ресурса.

Многозадачность и операционная система.

Большинство операционных систем (ОС) поддерживают многозадачность. Они последовательно переключают задачи одну на другую. В каждый момент времени процессор выполняет только одну задачу. В многопоточных процессорах одновременно могут выполняться несколько задач. ОС планирует какая из задач будет выполняться следующей, выбирает эту задачу и переключает контексты задач. Методы переключения зависят от стратегии, выбранной ОС.

Способы планирования.

1. Невытесняемое планирование. Новая задача переходит в состояние исполнения (Running), если предыдущая задача закончила свое выполнение или перешла в состояние ожидания (Waiting).

2. Полностью вытесняемое планирование. Задача вытесняется задачей с более высоким приоритетом. Такая ситуация может быть в двух случаях: пришла задача большего приоритета или приоритет текущей задачи понизился. Когда задача вытесняется она встает в очередь готовых задач соответствующего уровня приоритета. Заметьте, что вытесненная задача помещается не в конец, а в начало очереди. После завершения вытеснившей задачи вытесненная сможет отработать остаток своего времени.

3. Круговое планирование. Используется операция уступить новой задаче. Такое планирование может применяться для задач одного приоритета.

4. Завершение кванта. Каждой задаче выделяется квант времени. По окончании выполняется другая задача с текущим уровнем приоритета, если такой нет, то второй квант времени выполняется первая задача. Вытесненная задача ставится в очередь задач данного приоритета и переводится из состояния исполнения Running в состояние готовности (Ready). В Win у каждой задачи (потока) свое значение кванта. Это значение выражается не в единицах времени, а целым числом в так называемых квантовых единицах. Минимальный квант (6 единиц) по

умолчанию равен двум интервалам системного таймера. Максимальный соответствует 36 единиц. Длительный квант в ряде случаев позволяет свести к минимуму переключения контекста. Получая больший квант программы обслуживания клиентского запроса (в серверах) имеют больше шансов выполнить запрос и вернуться в состояние ожидания до истечения выделенного кванта. Длина временного интервала таймера зависит от аппаратной платформы и на большинстве однопроцессорных x86 систем составляет 10 мс, а на большинстве многопроцессорных систем - 15 мс. Величиной выделенного кванта можно управлять, в частности динамически приращивая квант при круговом планировании.

5. Адаптивное планирование. Основано на динамическом изменении приоритетов задач. По истечении выделенного кванта приоритет задачи снижается. Если с текущим уровнем приоритета нет задач, задача выполняется следующий квант, и приоритет снова снижается и так далее до базового. Если задача долгое время не выполнялась, то ее приоритет возрастает до тех пор, пока она не начнет выполняться. Техника назначения приоритетов и их изменения полностью определяется уровнем ОС.

6. Планирование по расписанию. Данное планирование предполагает использование комбинации разных способов.

Контекст. Переключение контекста.

Контекст – полная информация о текущем состоянии задачи и процессора, которая позволяет в любой момент продолжить вычисления. Контекст зависит от архитектуры процессора, типа задачи и работы ОС. В типичном случае полный контекст задачи включает следующие данные:

- счетчик команд,
- регистр состояния процессора (регистр флагов),
- содержимое остальных регистров процессора,
- указатели на стеки ядра ОС и пользователя,
- указатели на адресное пространство задачи (см. каталог таблиц страниц CR3 для Pentium),
- промежуточные результаты работы программы.

Переключение контекста позволяет загрузить в процессор из памяти новую задачу на выполнение, а информацию о старой сохранить. Объем контекста 32-разрядного процессора составляет 64 байта и более. Объем контекста RISC процессора значительно больше, так в процессоре Itanium число 64-разрядных регистров около 500. Ряд задач не требуют сохранения полного контекста, достаточно сохранить/ восстановить счетчик команд и регистр состояния процессора. Такими задачами могут быть задачи обслуживания прерываний клавиатуры, таймера и др. В этом случае размер контекста 8 байт (два слова).

Многозадачность в результате сводится к планированию прохождения задач, диспетчерированию и переключению контекста. Аппаратная поддержка возможна только операций по переключению контекста. Остальные слишком сложны и реализуются ОС.

Статическое и динамическое распределение ресурсов.

В мультипрограммной ЭВМ ресурсы могут распределяться как на статической, так и на динамической основе. В первом случае ресурсы распределяются до момента порождения процесса и являются для него постоянными. Освобождение ресурсов, занятых каким-либо процессом, происходит только в момент окончания этого процесса. При динамическом распределении ресурсы выделяются процессу по мере его развития.

Распределение на статической основе способствует наиболее быстрому развитию процессов в системе с момента их порождения. Распределение же ресурсов на динамической основе позволяет обеспечить эффективное использование ресурсов с точки зрения минимизации их простоев.

Схема статического распределения используется в том случае, когда необходимо гарантировать выполнение процесса с момента его порождения. В качестве недостатка этого подхода следует отметить возможность длительных задержек заявок на порождение процесса с момента поступления таких заявок в систему, так как необходимо ожидать освобождения всех требуемых заявке ресурсов и только при наличии их полного состава порождать процесс.

При динамическом распределении стремление уменьшить простои ресурсов приводит к увеличению сложности системы распределения ресурсов и, как следствие, к увеличению системных затрат на управление процессами. Поэтому необходим компромисс между сложностью алгоритмов планирования распределения ресурсов и эффективностью выполнения пакета задач.

Пример программы Мультизадачность (сокращенный).

```
#include <stdio.h>

struct ProcStats //struktura soderjashaya kontext zadachi
{   unsigned int rax, rbx, rcx, rdx; // 0  2  4  6
    unsigned int rsi, rdi, rbp, rsp; // 8 10 12 14
    unsigned int rcs, rds, res, rss; //16 18 20 22
    unsigned int rip, rflags, a, b;  //24 26 28 30
} Stats[3]; //massiv iz treh takih struktur

unsigned int current_proc; //nomer tekushey zadachi
unsigned int stats_offset; //adres konteksta zadachi
void interrupt (*oldHandler)(...); //ukazatel na staryi
obrabotchik preryvaniya
```

```
void interrupt IntHandler(...) //svoy obrabotchik
{    asm {
     mov  si, [current_proc]; //nomer procedury
     mov  cl, 5 //umnojaem nar 32 (razmer struktury)
     shl  si, cl
     //mov  ax, offset Stats
     mov  ax, [stats_offset] //pribavlyaem adres nachala
massiva struktur
     add  si, ax  //poluchaem adres nujnogo konteksta
     pop  ax      //zapisyvaem v nego registry
     mov  [si+12], ax  //bp
     pop  ax
     mov  [si+10], ax  //di
     pop  ax
     mov  [si+ 8], ax  //si
     pop  ax
     mov  [si+18], ax  //ds
     pop  ax
     mov  [si+20], ax  //es
     pop  ax
     mov  [si+ 6], ax  //dx
     pop  ax
     mov  [si+ 4], ax  //cx
     pop  ax
     mov  [si+ 2], ax  //bx
     pop  ax
     mov  [si+ 0], ax  //ax
     pop  ax
     mov  [si+24], ax  //ip
     pop  ax
     mov  [si+16], ax  //cs
     pop  ax
     mov  [si+26], ax  //flags
     mov  ax, sp
     mov  [si+14], ax  //sp
     mov  ax, ss
     mov  [si+22], ax  //ss
     mov  ax, [current_proc] //perehodim k sleduyushey zadache
     inc  ax
     cmp  ax, 3
     jb label1
     mov  ax, 0
    }
    label1:
    asm {
     mov  [current_proc], ax //vychislyaem adres eyo konteksta
     mov  si, ax
     mov  cl, 5
```

```
     shl  si, cl
     //mov  ax, offset Stats
     mov  ax, [stats_offset]
     add  si, ax       //zagrujaem registry
     mov  ax, [si+22]  //ss
     mov  ss, ax
     mov  ax, [si+14]  //sp
     mov  sp, ax
     mov  ax, [si+26]  //flags
     push ax
     mov  ax, [si+16]  //cs
     push ax
     mov  ax, [si+24]  //ip
     push ax
     mov  ax, [si+ 0]  //ax
     push ax
     mov  ax, [si+ 2]  //bx
     push ax
     mov  ax, [si+ 4]  //cx
     push ax
     mov  ax, [si+ 6]  //dx
     push ax
     mov  ax, [si+20]  //es
     push ax
     mov  ax, [si+18]  //ds
     push ax
     mov  ax, [si+ 8]  //si
     push ax
     mov  ax, [si+10]  //di
     push ax
     mov  ax, [si+12]  //bp
     push ax
    }
    oldHandler(); //vyzyvaem staryi obrabotchik
}
void Proc1(void); void Proc2(void); void Proc3(void);

int main(void)
{unsigned int i, rd, rc, re, rs, ri0, ri1, ri2, f;
    for(i=0;i<3;i++)
     memset(&Stats[i],0,sizeof(ProcStats));
    asm {   //zapisyvaem registry vo vremennye peremennye
     mov  ax, cs
     mov  [rc], ax
     mov  ax, ds
     mov  [rd], ax
     mov  ax, es
     mov  [re], ax
     mov  ax, ss
```

```
     mov  [rs], ax
     mov  [ri0], offset Proc1 //adresa nachala procedur
     mov  [ri1], offset Proc2
     mov  [ri2], offset Proc3
     pushf
     pop  ax
     mov  [f], ax
    }
    for(i=0;i<3;i++) //zapisyvaem znacheniya segmentnyh
registrov,
    {                //flagov
     Stats[i].rcs = rc;  //i adresa nacha procedur
     Stats[i].rds = rd;  //v sootvetstvuyushye konteksty
     Stats[i].res = re;
     Stats[i].rss = rs;
     Stats[i].rflags = f;
    }
    Stats[0].rip = ri0;
    Stats[1].rip = ri1;
    Stats[2].rip = ri2;
    current_proc = 0; //tekushaya procedura - pervaya
    stats_offset = (unsigned)&Stats; //adres nachala massiva
    oldHandler = getvect(0x9); //perekluchaemsya po najatiyu
klavishy
    setvect(0x9,IntHandler);
    Proc1();
    setvect(0x9,oldHandler);
    return 0;
}
void Proc1(void)
{    while(1)
     printf("Proc1 is working %d\n",current_proc);   }
----------------------------
```

Темы итоговых индивидуальных заданий

Задания первого этапа изучения Ассемблера тема “Работа с текстом”

1. Упорядочить список группы из 10 человек, состоящий из строк " ФАМИЛИЯ И. О." по возрастанию инициала имени. Выведите на экран начальный и конечный список.

2. Дан список из 20 слов по 10 символов в каждом. Напечатать его в обратном алфавитном порядке, предварительно удалив из него повторяющиеся слова. При сортировке игнорировать высоту букв (Например, A = a).

3. Разработать набор процедур работы с очередью, а именно:

- включение нового элемента;

- выборка очередного элемента (со сдвигом очереди)

Разработать демо-программу.

4. В файле хранится каталог файлов в формате команды DIR. Разработать программу переупорядочения каталога

- по имени,
- по расширениям,
- по размерам,

а также программу выдачи его на экран.

5. Дано арифметическое выражение. Разработать программу проверки правильности по следующим критериям:

- допустимые знаки операций ("+", "-", "*","/");
- допустимые константы (целые без знака);
- допустимые переменные (до пяти латинских букв);
- скобки (только круглые, они должны быть парными).

Продумать диагностику по разным типам ошибок.

6. Разработать игру: кто придумает больше слов из символов заданной строки. Проверять:

- на допустимость по символам;
- по словарю, есть ли такие слова.

Предусмотреть возможность дополнения словаря и игнорирование высоты букв (A = a).

7. Разработать программу поиска модели в тексте (см. любой текстовый редактор).

8. Разработать процедуру слияния двух упорядоченных списков вида "ФАМИЛИЯ И. О." с сохранением алфавитного порядка. Исходные и конечный списки вывести на экран.

9. Разработать процедуру, показывающую в зависимости от указания пользователя либо первые 15, либо последние 15 строк текстового файла. При переключении очищать экран.

10. Организовать печатание произвольного текстового файла на экране. Вертикальный размер окна (количество строк) может меняться. Для печатания желательно использовать PgUP и PgDOWN.

11.Составить "электронную зачетную ведомость" для вашей группы по ТТО:

N ¦Фамилия И.О.¦Номер u1079 зачет. книжки ¦Отметка о зачете¦Дата п/п¦ ¦ ¦ ¦

Предусмотреть возможность включения слова "зачет" и даты получения зачета.

12. Составить программу построения диаграммы повторяемости букв в тексте. Формат произвольный. Использовать для текста:

"Fools! You have no perception!
The stakes yoa are gambling
Are frighteninly high!
We must crush him comletely
So like John before him
This Jesus must die!"

13. Составить программу склонения мужской фамилии в родительный падеж. Использовать для организации диалог вида:

Есть Вишневский?

Нет Вишневского.

14. Составить программу склонения женской фамилии в родительный падеж. Использовать для организации диалог (ввод и вывод текста) вида:

Есть Шевченко?

Нет Шевченко.

15. Написать программу для преобразования целого числа в римские цифры. Проверить в пределах 1-100. Число вводится в десятичной системе, а выводится в римской.

16. Написать программу, записывающую однобайтовое целое число без знака прописью, например, 128 = сто двадцать восемь

17. В списке людей "ФАМИЛИЯ ИМЯ ОТЧЕСТВО" необходимо найти всех людей с заданным именем. Выводить на экран в виде:

ИМЯ ..... количество повторений

18. Из текст выбрать слова, начинающиеся буквой из первой половины алфавита.

Образец текста:

The end is just a little harder
When brought about by friends!
For all who cares this wine could be my blood!
For all who cares this bread could be my body!
The end! This is my blood you drink, this is my body you eat!

19. Составить программу повторяемости слов в тексте. Использовать для текста:

See my eyes I can hardly see
See me stand I can hardly walk
I believe you can make me whole
See my tongue I can hardly talk
See my skin I'm a mass of blood
See my legs I can hardly stand
I believe you can make me well
See my purse I'm a poor poor man.

20. Разработать программу, которая выводит строки текста в указанном с клавиатуры порядке.

Задания второго этапа изучения Ассемблера темы "Таймер", "Клавиатура", "Прерывания", "Звук", "Отладка", "Мультизадачность".

1. Программа "Будильник1". Задайте с клавиатуры время относительно текущего времени (например, 2 минуты). Перехватите прерывание клавиатуры и при повторном наборе (нажатии клавиш). Ваша программа сообщает "Ошибка". При наступлении заданного времени выдайте звуковой сигнал с использованием 2-го канала таймера. Звук должен быть многоголосый.

2. Программа "Будильник2". Будильник должен прозвенеть в абсолютное время (11:45). Перехватите прерывание клавиатуры и при повторном наборе (нажатии клавиш). Ваша программа u1089 сообщает "Ошибка". При наступлении заданного времени выдайте звуковой сигнал.

3. Программа "Будильник3". Задайте с клавиатуры время относительно текущего времени (например, 2 минуты). На экран выводится время, оставшееся до звонка. При наступлении заданного времени выдайте звуковой сигнал.

4. Программа "Будильник4". Задайте с клавиатуры время относительно текущего времени (например, 2 минуты). Перехватите прерывание клавиатуры (номер 9) и при повторном наборе (нажатии клавиш). Ваша программа сообщает "Ошибка". При наступлении заданного времени выдайте звуковой сигнал. Определите свободные векторы внешних прерываний, выведите на экран их количество.

5. Определите экспериментально время обработки прерывания от канала 0 таймера. Напишите программу, которая перехватывает прерывание таймера и обеспечивает мигание индикаторов клавиатуры через каждую 1 сек.

6. Напишите программу для измерения времени выполнения одной тестовой подпрограммы тремя способами с разной точностью (используя ячейку 46C, повышение частоты прерываний системного времени, нулевой канал таймера).

Изменение частоты прерываний от канала 0 исказит значение системного времени. Исключите это искажение. Распечатайте содержимое ячейки, используемой для счета импульсов времени 0046C,D,E,F.

7. Подмена клавиш 1. Замените клавишу A на B. Напишите программу, обработки прерывния клавиатуры, которая заменяет скан-коды клавиш на одно системное прерывание. Замедлите системные часы в 100 раз путем прореживания прерываний таймера.

8. Подмена клавиш 2. Замените клавишу F1 на F2. Напишите программу, обработки прерывния клавиатуры, которая заменяет скан-коды

клавиш. При каждом нажатии выдавайте звуковой сигнал с использованием 2-го канала таймера.

9. Подмена клавиш 3. Замените клавишу Del на Ins. Напишите программу, обработки прерывания клавиатуры, которая заменяет скан-коды клавиш. При каждом нажатии выдавайте звуковой сигнал с использованием порта 061h.

10. Работа с клавиатурой 1. Напишите программу, обработки прерывания клавиатуры, которая выводит на экран скан-код и код, который соответствует нажатой клавише из буфера клавиатуры.

11. Работа с клавиатурой 2. Напишите программу работы с портом клавиатуры, который используется для чтения скан-кодов и для управления клавиатурой. Выведите на экран скан-коды нажатой клавиши, выполните функции управления клавиатурой.

12. Отладчик 1. Постройте программу, которая содержит два фрагмента, один из которых играет роль отладчика, а второй имитирует отлаживаемую программу. Используйте режим трассировки (флаг T). По нажатию кнопки отладчик должен распечатывать содержимое регистра и обеспечивать мигание индикаторов клавиатуры (или выдавать звуковой сигнал).

13. Отладчик 2. Постройте программу, которая содержит два фрагмента, один из которых играет роль отладчика, а второй имитирует отлаживаемую программу. Используйте режим трассировки (флаг T). Имитатор должен обеспечивать вывод на экран адрес отлаживаемой программы и звуковой сигнал с использованием 2-го канала таймера.

14. Определите свободные векторы внешних прерываний, выведите их на экран. По одному из векторов инициируйте прерывание и выдайте сигнал с использованием 2-го канала таймера

15. Мультизадачность 1. Организуйте мультизадачную среду с выводом на экран номера задачи . Переключение происходит по нажатию клавиши мышки.

16. Мультизадачность 2. Организуйте мультизадачную среду (3 задачи) с выводом на экран номера задачи. Переключение происходит по алгоритму кругового планирования с выделенными квантами времени.

17. Мультизадачность 3. Организуйте мультизадачную среду (3 и более задач) с выводом на экран номера задачи. Переключение происходит по алгоритму адаптивного планирования.

18. Музыкальный инструмент1. Напишите программу, в которой имитируется музыкальный инструмент. При нажатии на кнопки клавиатуры выводится звуковой сигнал, высота которого зависит от нажатой кнопки (используйте 2-й канал таймера).

19. Музыкальный инструмент2. Напишите программу, в которой имитируется музыкальный инструмент. Запоминается последовательность

нажатия на кнопки клавиатуры и затем проигрывается звуковой фрагмент. Мелодия зависит от последовательности нажатых кнопок.

20. Музыкальный инструмент3. Напишите программу, в которой имитируется музыкальный инструмент. При нажатии на кнопки клавиатуры выводится звуковой сигнал, высота которого зависит от нажатой кнопки (используйте прерывания BIOS).

Примеры заданий по кодированию команд и данных

Add [r11+r12], rbx 00ff F00f shr rcr sar 2,2	Mov [r14+4*r12], rbx 000f 800f shr, rcr, sar 12,5	And [r11+rax+4], dl 010f F00f shr, sar, rcr 0,4

Заключение

В заключение можно отметить, что среди вопросов низкоуровневого программирования в данном цикле лабораторных работ рассмотрено программирование ЭВМ на базе процессоров Pentium с использованием Ассемблера и Си. В виду разнообразия платформ и малой распространенности чистого программирования на Ассемблере изучению Ассемблера отводится не значительное время. Однако этого времени достаточно для получения основных знаний и навыков программирования. Основное внимание при программировании уделено интеграции Ассемблера и Си.

В качестве дальнейшего направления изучения низкоуровневого программирования можно определить следующие темы лабораторных работ:

1. Дизассемблирование программ.
2. Многозадачная операционная система.
3. Создание собственного загрузчика.
4. Ассемблер под Linux. Использование функций из библиотеки GLIBC - стандартная библиотека Си от GNU.
5. Ассемблер под Windows. FASM

Библиографический список

1. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М.: «ДИАЛОГ-МИФИ»,2005.- 288с.
2. Дао Л. Программирование микропроцессора 8088.-М.:Мир,1988
3. Юров В.И. Assembler: Учебник для вузов. — 2-е изд. — СПб.: Питер, 2010. — 637 с.
4. Аблязов Р. – Программирование на ассемблере на платформе x86-64, ДМК Пресс, 2016, 304с.
5. Intel® 64 and IA-32 Architectures Software Developer’s Manual
6. Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000.- Спб.:Питер, 2001, 752с.

Электронные ресурсы

7. Молодяков С.А. ЭВМ и периферийные устройства [Электронный ресурс]: учебное пособие. Ч. 1 . Основы организации ЭВМ; СПб., 2012.- http://elib.spbstu.ru/dl/2759.pdf/download/2759.pdf.
8. Молодяков С.А.. Методические указания к выполнению курсовой работы по дисциплине "ЭВМ и периферийные устройства" [Электронный ресурс] : [учебное пособие] .— СПб., 2012.- http://elib.spbstu.ru/dl/2760.pdf/download/2760.pdf
9. http://programmersclub.ru/assembler/
11. ru.wikipedia.org›wiki/Ассемблер
12. assmbler.ru
13. http://www.codenet.ru/
14. http://www.stolyarov.info/books/pdf/nasm_unix.pdf

Молодяков Сергей Александрович,
Петров Александр Владимирович

АРХИТЕКТУРА ЭВМ. НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум

Налоговая льгота – Общероссийский классификатор продукции
ОК 005-93, т. 2; 953005 – учебная литература

Подписано к печати XX.XX.2019. Формат 60x84/16. Печать цифровая.
Усл. печ. л. 11,75. Тираж 80 экз. Заказ

Отпечатано с готового оригинал-макета, предоставленного авторами,
в типографии Издательства Политехнического университета.
195251, Санкт-Петербург, Политехническая ул., 29.
Тел.: (812) 550-40-14
Тел./факс: (812) 297-57-76