Надо сделать: Получить оценку

Цели:

1. закрепить знания о регистрах общего назначения 32-разрядных процессоров INTEL;
2. научиться использовать косвенную адресацию для работы с оперативной памятью;
3. научиться использовать команды умножения и деления целых чисел.

Основная нагрузка при работе компьютера ложится на процессор и память. Процессор выполняет команды, хранящиеся в памяти. В памяти хранятся также и данные. Между процессором и памятью происходит непрерывный обмен информацией. Процессор имеет свою небольшую память, состоящую из **регистров**. Команда процессора, использующая находящиеся в регистрах данные, выполняется много быстрее аналогичных команд над данными в памяти. Поэтому часто для того, чтобы выполнить какую-либо команду, данные для неё предварительно помещают в регистры. Результат команды можно при необходимости поместить обратно в память. Обмен данными между памятью и регистрами осуществляют **команды пересылки**. Кроме этого, можно обмениваться данными между регистрами, посылать и получать данные от внешних устройств. В регистр и ячейку памяти можно посылать и непосредственный **операнд** – число. Кроме этого имеются команды, с помощью которых можно помещать и извлекать данные из **стека** – специальной области памяти, используемой для хранения адресов возврата из функций, передаваемых в функцию параметров и локальных переменных.

Адресация и выделение памяти

Для процессора вся память представляет собой последовательность однобайтовых ячеек, каждая из которых имеет свой адрес. Для того, чтобы оперировать большими числами, пары ячеек объединяют в слова, пары слов – в двойные слова, пары двойных слов – в учетверенные слова. Чаще всего в программах оперируют **байтами**, **словами** и **двойными словами** (в соответствии с одно-, двух- и четырехбайтовыми регистрами процессоров). **Адресом слова и двойного слова является адрес их младшего байта.**

На листинге 1 представлен пример доступа к памяти при помощи косвенной адресации. Рассмотрим подробно. Прежде всего, отметим, что в программу включен заголовочный файл **<windows.h>**, который содержит заголовки всех основных API-функций ОС Windows, а также определение большого количества структур, типов переменных (в частности, определение типа **DWORD**, который сводится просто к **unsigned int**). В ассемблерных командах используются переменные, определенные средствами языка Си. Это связано с тем, что встроенный в Си ассемблер не позволяет осуществлять резервирование памяти. Адресация памяти с помощью переменных называют также **прямой адресацией**. **Косвенная адресация** состоит в следующем. Если адрес ячейки содержится в регистре, например, **EAX**, то для того, чтобы послать туда число 100, нужно написать **MOV BYTE PTR [EAX], 100**. Префикс **BYTE PTR** указывает, что в операции участвует однобайтовая ячейка памяти (можно использовать **WORD PTR**, **DWORD PTR** – это будет соответствовать двух- и четырехбайтовому операнду). Чтобы получить адрес ячейки памяти, используется команда **LEA**.

```
/* использование косвенной адресации */
/* подключаемые заголовочные файлы */
#include <stdio.h> // необходим для работы printf
#include <conio.h> // необходим для работы _getch();
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;

/* глобальные переменные */
BYTE  a=10; // 8-битное беззнаковое целое число
DWORD addressRet; // переменная для хранения адреса

/* главная функция */
void main()
{
	__asm {
		LEA EAX, a;	// загрузка эффективного адреса переменной a в регистр EAX
			// (в 32-разрядной ОС адрес ячейки памяти занимает 4 байта,
			// поэтому для хранения адреса надо использовать расширенные
			// регистры)

		MOV addressRet, EAX; // помещаем в переменную addressRet адрес переменной
				// a, хранящийся в регистре EAX. Обратите внимание:
				// этот адрес меняется при каждом запуске программы

		MOV BYTE PTR [EAX], 100; // помещаем по адресу, хранящемуся в регистре EAX
				// число 100 - фактически, присваиваем переменной a
				// значение  100
	};
	printf("address of variable a is %u\n", addressRet); // выводим адрес переменной a
	printf("value of variable a = %u\n", a); // выводим значение переменной a
	_getch();
}
```

Здесь используется доступ к переменной типа **BYTE** по указателю – структура **BYTE PTR [EAX]**. Немного позже мы увидим, как этот прием используется при написании программ.

Задания.

- Попробуйте записать по адресу переменной **a**, хранящемуся в регистре **EAX**, число **260**. Какой ответ вы получили? Почему?
- Задайте переменную **b** типа **WORD** и переменную **c** типа **DWORD**. Используя косвенную адресацию, запишите в эти переменные числа **1023** и **70000**, соответственно.
- Поместите в переменную **c** число **70000**, используя указатель типа **BYTE**:

```
LEA EAX, c;
MOV BYTE PTR [EAX], 70000;
```

Объясните полученный результат (напоминаем, что адресом слова или двойного слова является адрес их младшего байта). Проделайте то же самое, используя указатель типа WORD**.**

На листинге 2 представлена программа, иллюстрирующая способы доступа к переменным по указателям. Наберите эту программу. Разберитесь с комментариями. Попробуйте поменять элементы массива. Попробуйте выводить результаты в шестнадцатеричной системе (вместо **%u** в строке формата функции **printf()** используйте **%x**).

```
/* использование косвенной адресации */
#include <stdio.h> // необходим для работы printf
#include <conio.h> // необходим для работы _getch();
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;
BYTE ar[6] = {1, 12, 128, 50, 200, 10}; // статический массив типа BYTE
BYTE  a1, a2, a3, a4, a5; // 8-битные беззнаковые числа
WORD  b1, b2;          // 16-битные беззнаковые числа
DWORD c;               // 32-битное беззнаковое число

void main()
{
        __asm {
                LEA EBX, ar; // загрузка эффективного адреса первого элемента массива
                        // ar в регистр EAX
                MOV AL, BYTE PTR [EBX]; // помещаем в регистр AL число (типа BYTE)
                                // число, записанное по адресу, хранящемуся
                                // в регистре EBX, то есть первый элемент массива
                MOV a1, AL; // записываем содержимое регистра AL в переменную a

                /*помещаем в переменную a2 число, записанное по адресу "начало массива
                плюс 1 байт", то есть по адресу второго элемента массива*/
                MOV AL, BYTE PTR [EBX] + 1;
                MOV a2, AL;
                /*помещаем в переменную a3 число, записанное по адресу "число, записанное
в
                регистре EBX плюс 1", то есть по адресу второго элемента массива*/
                MOV AL, BYTE PTR [EBX+1] ;
                MOV a3, AL;
                /*помещаем в переменную a4 число, записанное по адресу "номер,
                хранящийся в регистре EDX, начиная с номера, записанного регистре EBX",
                то есть второй элемент массива*/
                MOV EDX, 1;
                MOV AL, BYTE PTR [EBX][EDX];
                MOV a4, AL;

                записанных в регистрах EBX и EDX", то есть второй элемент массива*/
                MOV AL, BYTE PTR [EBX+EDX];
                MOV a5, AL;
                /*помещаем в переменную b1 2 и 1 элементы массива*/
                MOV AX, WORD PTR [EBX];
                MOV b1, AX;
                /*помещаем в переменную b2 4 и 3 элементы массива*/
                MOV AX, WORD PTR [EBX]+2;
                MOV b2, AX;
                /*помещаем в переменную c 6, 5, 4 и 3 элементы массива*/
                MOV EAX, DWORD PTR [EBX]+2;
                MOV c, EAX;

        };

        printf("first element of array a1 = %u \n", a1);
        printf("second element of array a2 = %u \n", a2);
        printf("second element of array (another way) a3 = %u \n", a3);
        printf("second element of array (base addressation) a4 = %u \n", a4);
        printf("second element of array (base addr. - another way) a5 = %u \n", a5);
        printf("1, 2 elements of array  b1 = %u \n", b1);
        printf("3, 4 elements of array  b2 = %u \n", b2);
        printf("3, 4, 5, 6 elements of array  c = %u \n", c);
        _getch();
}
```

Доступ к переменной по указателю используется и в языках высокого уровня (очень часто – при создании динамических массивов).

Указатель – это переменная, которая содержит адрес другой переменной (говорят, что указатель указывает на переменную того типа, адрес которой он содержит). Существует одноместная (унарная, т.е. для одного операнда) **операция взятия адреса переменной** & (амперсанд, как в названии мультфильма Tom&Jerry). Если имеем объявление **int a**, то можно определить **адрес** этой переменной: **&a**. Если **Pa** – указатель, который будет указывать на переменную типа **int**, то можно записать: **Pa=&a**. Существует унарная операция ***** (она называется **операцией**

разыменования), которая действует на переменную, содержащую адрес объекта, т.е. на указатель. При этом извлекается содержимое переменной, адрес которой находится в указателе. Если **Pa=&a**, то, воздействуя на обе части операцией * получим (по определению этой операции): ***Pa=a**. Исходя из этого, указатель объявляется так:

<тип переменной> * <имя указателя>

Это и есть правило объявления указателя: указатель на переменную какого-то типа – это такая переменная, при воздействии на которую операцией разыменования получаем значение переменной того же типа. На листинге 3 приведен пример использования указателя в языке Си.

Один из наиболее часто встречающихся случаев – использование указателей для **динамического выделения памяти при создании массивов** (листинг 5).

```
/* динамическое выделение памяти */
#include <stdio.h> // необходим для работы printf
#include <conio.h> // необходим для работы _getch()
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD
#include <stdlib.h> // необходим для работы malloc()
#include <malloc.h> // необходим для работы malloc()

void main()
{
   int* ptint; // указатель на переменную типа int
/* Выделяем память под массив. Аргумент функции malloc() - число байт.
Нам нужен массив из 10 целых чисел. Поэтому общее число байт - размер числа
типа int (определяется функцией sizeof()), умноженный на число элементов массива.
Стоящая перед malloc() конструкция (int*) осуществляет приведение к типу int*
(то есть теперь выделенная память будет рассматриваться компилятором как
совокупность 4 байтных ячеек, в которых хранятся числа типа int) */
   ptint = (int*)malloc(10 * sizeof(int));
         /* заполняем массив */
         for(int i=0; i<10; i++)
                  ptint[i]=i;
         /*выводим элементы массива*/
         for(int i=0; i<10; i++)
                  printf("%d ",ptint[i]);
free(ptint); // освобождаем память
         _getch();
}
```

```
/* получение адреса переменной - сравнение C и Assembler */
#include <stdio.h> // необходим для работы printf
#include <conio.h> // необходим для работы _getch();
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;

BYTE a=10;
BYTE *cAddr;
DWORD asmAddr;
BYTE b;

void main()
{

         __asm {
                  LEA EBX, a;     // загрузка эффективного адреса переменной a в регистр EBX
                  MOV asmAddr, EBX; // помещаем в переменную asmAddr содержимое регистра
EBX,
                           // т.е. адрес переменной a
         };

         cAddr=&a; // записываем в переменную типа BYTE* адрес переменной типа BYTE
         b=*cAddr; // осуществляем разыменование указателя на переменную a

         printf("Assembler: address of a is %u\n", asmAddr);
         printf("C: address of a is %u\n", cAddr);
         printf("C: value of a is %u\n", b);
         _getch();
}
```

На листинге 4 представлена программа, позволяющая получать адреса элементов массивов разных типов средствами Си. Обратите внимание на значения соседних адресов элементов массива.

```
/* адресация в массивах */
#include <stdio.h> // необходим для работы printf
#include <conio.h> // необходим для работы _getch();
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;

unsigned int mas[4]; // массив 4-байтовых целых чисел
unsigned int *ptrMas; // указатель на переменную типа unsigned int
unsigned short int masShort[4]; // массив 2-байтовых целых чисел
unsigned short int *ptrMasShort; // указатель на переменную типа unsigned short int
BYTE masBYTE[4]; // массив 1-байтовых целых чисел
BYTE *ptrMasBYTE; // указатель на переменную типа BYTE

void main()
{
	ptrMas = mas; // помещаем в указатель адрес первого элемента массива
	ptrMasShort = masShort;
	ptrMasBYTE = masBYTE;

	printf("array of int \n");
	for(int i=0; i<4; i++)
		printf("int pointer+%u = %u\n", i, ptrMas+i);

	printf("\narray of short int \n");
	for(int i=0; i<4; i++)
		printf("short pointer+%u = %u\n", i, ptrMasShort+i);

	printf("\narray of BYTE \n");
	for(int i=0; i<4; i++)
		printf("byte pointer+%u = %u\n", i, ptrMasBYTE+i);
	_getch();
}
```

Один из наиболее часто встречающихся случаев – использование указателей для **динамического выделения памяти при создании массивов** (листинг 5).

```
/* динамическое выделение памяти */

#include <stdio.h> // необходим для работы printf

#include <conio.h> // необходим для работы _getch()

#include <windows.h> // содержит определение типов BYTE, WORD, DWORD

#include <stdlib.h> // необходим для работы malloc()

#include <malloc.h> // необходим для работы malloc()

 void main()

{

   int* ptint; // указатель на переменную типа int

/* Выделяем память под массив. Аргумент функции malloc() - число байт.

Нам нужен массив из 10 целых чисел. Поэтому общее число байт - размер числа

типа int (определяется функцией sizeof()), умноженный на число элементов массива.

Стоящая перед malloc() конструкция (int*) осуществляет приведение к типу int*

(то есть теперь выделенная память будет рассматриваться компилятором как

совокупность 4 байтных ячеек, в которых хранятся числа типа int) */

   ptint = (int*)malloc(10 * sizeof(int));

         /* заполняем массив */

        for(int i=0; i<10; i++)
```

```
        ptint[i]=i;

        /*выводим элементы массива*/

        for(int i=0; i<10; i++)

        printf("%d ",ptint[i]);

free(ptint); // освобождаем память

        getch();

}
```

Задание. Выведите на экран адреса элементов массива, созданного в программе, показанной на листинге 5. Попробуйте создать динамический массив типа **double**, заполнить его, вывести на печать элементы массива и их адреса.

Арифметические операции над целыми числами

Сложение и вычитание целых чисел

Рассмотрим 3 основные команды сложения. Команда **INC** осуществляет инкремент, т.е. увеличение содержимого операнда на 1, например, **INC EAX**. Команда **INC** устанавливает флаги **OF, SF, ZF, AF, PF** в зависимости от результатов сложения. Команда **ADD** осуществляет сложение двух операндов. **Результат пишется в первый операнд (приемник)**. Первый операнд может быть регистром или переменной. Второй операнд – регистром, переменной или числом. Невозможно, однако, осуществлять операцию сложения одновременно над двумя переменными. Команда действует на флаги **CF, OF, SF, ZF, AF, PF**. Её можно использовать для знаковых и для беззнаковых чисел. Команда **ADC** осуществляет сложение двух операндов подобно команде **ADD** и флага (бита) переноса. С её помощью можно осуществлять сложение чисел, размер которых превышает 32 бита или изначально длина операндов превышает 32 бита.

```
/* сложение целых чисел */
#include <stdio.h> // необходим для работы printf()
#include <conio.h> // необходим для работы _getch()
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;
int a,b,c; DWORD d,e,f,m,n,l,k;
void main()
{
        a=100; b=-200; f=0;
        d=0xffffffff;      e=0x00000010;
        m=0x12345678; n=0xeeeeeeee; l=0x11111111; k=0x22222222;
        __asm{
    /* сложение положительного и отрицательного чисел */
                MOV EAX, a;
                ADD EAX, b;
                MOV c, EAX;
        /* сложение двух больших чисел */
                MOV EAX, e; // EAX = 0x00000010
                ADD d, EAX; // результат превышает 4 байта, поэтому флаг CF
                        // устанавливается в 1:
                        //   0xffffffff
                        // + 0x00000010
                        //   ---------
                        //   0x0000000f (и 1 должна переноситься в следующий разряд,
                        //   но его нет, поэтому устанавливается флаг CF)
                ADC f, 0; // осуществляет сложение двух операндов (подобно команде ADD) и
                        // флага (бита) переноса CF. Вначале f=0, второй операнд также 0,
                 // поэтому в данном случае выполнение команды сводится к помещению в
                      // переменную f значения CF
        /* сложение двух больших чисел, расположенных в паре регистров */
                MOV EDX, m; // поместили в EDX старшие 4 байта первого числа,
               //EDX=0x12345678
                MOV EAX, n; // поместили в EAX младшие 4 байта первого числа,
               // EAX=0xeeeeeeee
                MOV ECX, l; // поместили в ECX старшие 4 байта второго числа,
               // ECX=0x11111111
                MOV EBX, k; // поместили в EBX младшие 4 байта первого числа,
               // EBX=0x22222222
                ADD EAX, EBX; // сложили младшие 4 байта
                MOV n, EAX;
                ADC EDX, ECX; // сложили старшие 4 байта
                MOV m, EDX;
        };
        printf("c=a+b=%d\n",c);
        printf("f=d+e=%x%x\n",f,d);
        printf("sum of lowest 4 bytes = %x\n",n);
        printf("sum of highest 4 bytes = %x\n",m);
  _getch();
```

```
/*вычитание целых чисел*/
#include <stdio.h> // необходим для работы printf()
#include <conio.h> // необходим для работы _getch()
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;
int a,b,c;
__int64 i,j,k;
void main()
{
        a=100; b=-200;
        i=0x1ffffffff;
        j=0x1fffffffb;
        __asm{
     /* вычитание 32-битных чисел */
                MOV EAX, a;
                SUB EAX, b;
                MOV c, EAX;
                /* вычитание 64-битных чисел */
                MOV EAX, DWORD PTR i; // поместили в EAX адрес младших 4 байт числа i.
                        // По этому адресу записано число 0xffffffff
                MOV EDX, DWORD PTR i+4; // поместили в EDX адрес старших 4 байт числа i.
                        // По этому адресу записано число 0x00000001
                MOV EBX, DWORD PTR j; // поместили в EBX адрес младших 4 байт числа j.
                        // По этому адресу записано число 0xfffffffb
                MOV ECX, DWORD PTR j+4; // поместили в ECX адрес старших 4 байт числа j.
                        // По этому адресу записано число 0x00000001
                SUB EAX, EBX; // вычитаем из младших 4 байт числа i младшие 4 байта
                        // числа j. Эта операция влияет на флаг CF
                SBB EDX, ECX; // вычитаем из старших 4 байт числа i старшие 4 байта
                        // числа j, а также флаг CF
                MOV DWORD PTR k, EAX; // помещаем в память младшие 4 байта результата
                MOV DWORD PTR k+4, EDX; // помещаем в память старшие 4 байта результата
        };
        printf("c=a+b=%d\n",c);
        printf("k=i-j=%I64x\n",k);// интерпретируем выводимое число как __int64
  _getch();
}
```

Умножение целых чисел

В отличие от сложения и вычитания **умножение** чувствительно к знаку числа, поэтому существует две команды умножения: **MUL** – для умножения **беззнаковых** чисел, **IMUL** – для умножения **чисел со знаком**. Единственным оператором команды **MUL** может быть регистр или переменная. Здесь важен размер этого операнда (источника).

- Если операнд **однобайтовый**, то он будет умножаться на **AL**, соответственно, результат будет помещен в регистр **AX** независимо от того, превосходит он один байт или нет. Если результат не превышает 1 байт, то флаги **OF** и **CF** будут равны 0, в противном случае – 1.
- Если операнд **двухбайтовый**, то он будет умножаться на **AX**, и результат будет помещен в пару регистров **DX:AX** (а не в **EAX**, как могло бы показаться логичным). Соответственно, если результат поместится целиком в **AX**, т.е. содержимое **DX** будет равно 0, то нулю будут равны и флаги **CF** и **OF**.
- Наконец, если оператор-источник будет иметь длину **четыре байта**, то он будет умножаться на **EAX**, а результат должен быть помещен в пару регистров **EDX:EAX**. Если содержимое **EDX** после умножения окажется равным нулю, то нулевое значение будет и у флагов **CF** и **OF**.

Команда **IMUL** имеет 3 различных формата. Первый формат аналогичен команде **MUL**. Остановимся на двух других форматах.

IMUL operand1, operand2

operand1 должен быть регистр, **operand2** может быть числом, регистром или переменной. В результате выполнения умножения (**operand1** умножается на **operand2**, и результат помещается в **operand1**) может получиться число, не помещающееся в приемнике. В этом случае флаги **CF** и **AF** будут равны 1 (0 в противном случае).

IMUL operand1, operand2, operand3

В данном случае **operand2** (регистр или переменная) умножается на **operand3** (число) и результат заносится в **operand1** (регистр). Если при умножении возникнет переполнение, т.е. результат не поместится в приемник, то будут установлены флаги **CF** и **OF**. Применение команд умножения приведено на листинге 8.

```
#include <stdio.h> // необходим для работы printf()
#include <conio.h> // необходим для работы _getch()
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;
DWORD a=100000; __int64 b;
int c=-1000; int e;
void main()
{
        __asm{
    /* беззнаковое умножение */
                MOV EAX, 100000; // поместили в EAX число, превышающее 2 байта
                MUL DWORD PTR a; // умножаем содержимое регистра EAX на a,
                        // результат будет помещен в пару регистров
                        // EDX:EAX
                MOV DWORD PTR b, EAX; // помещаем в младшие 4 байта
                        // 8-байтной переменной b младшие 4 байта результата
                MOV DWORD PTR b+4, EDX; // помещаем в старшие 4 байта
                        // 8-байтной переменной b старшие 4 байта результата
        /* знаковое умножение */
                IMUL EAX, c, 1000; // умножаем с на 1000 и результат помещаем в EAX
                MOV e, EAX; // помещаем результат умножения в переменную e
        };
        printf("a*100000 = %I64d\n",b);// интерпретируем выводимое число как __int64
        printf("e = %d\n",e);
  _getch();
}
```

Деление целых чисел

Деление беззнаковых чисел осуществляется с помощью команды **DIV**. Команда имеет только один операнд – это делитель. Делитель может быть регистром или ячейкой памяти. В зависимости от размера делителя выбирается и делимое.

- Делитель имеет размер **1 байт**. В этом случае делимое помещается в регистре **AX**. Результат деления (частное) содержится в регистре **AL**, в регистре **AH** будет остаток от деления.

- Делитель имеет размер **2 байта**. В этом случае делимое помещается в паре регистров **DX:AX**. Результат деления (частное) содержится в регистре **AX**, в регистре **DX** будет остаток от деления.
- Делитель имеет размер **4 байта**. В этом случае делимое помещается в паре регистров **EDX:EAX**. Результат деления (частное) содержится в регистре **EAX**, в регистре **EDX** будет остаток от деления.

Команда знакового деления **IDIV** полностью аналогична команде **DIV**. Существенно, что для команд деления значения флагов арифметических операций не определены. В результате деления может возникнуть либо переполнение, либо деление на 0. Обработку исключения должна обеспечить операционная система.

```
#include <stdio.h> // необходим для работы printf()
#include <conio.h> // необходим для работы _getch()
#include <windows.h> // содержит определение типов BYTE, WORD, DWORD;
DWORD a,b,c;
void main()
{
        a=100000; // делимое -  4 байта
        __asm{
    /* беззнаковое деление */
                MOV EAX, a; // поместили младшие 4 байта делимого в регистр EAX
                MOV EDX, 0; // поместили старшие 4 байта делимого в регистр EDX
                MOV EBX, 30; // поместили в EBX делитель (4 байта!)
                DIV EBX; // выполнили деление содержимого EDX:EAX на
                      // содержимое EBX
                MOV b, EAX; // помещаем в b частное
                MOV c, EDX; // помещаем в c остаток
        };
        printf("div b = %d\n",b);
        printf("mod c = %d\n",c);
  _getch();
}
```

Добавить ответ на задание

Состояние ответа

Состояние ответа на задание	Ответы на задание еще не представлены
Состояние оценивания	Не оценено