МИНИСТЕРСТВО ТРАНСПОРТА РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

(РУТ (МИИТ)

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

Отчет по лабораторной работе №4

«Логические команды и команды манипулирования битами»

по дисциплине

«Машинно-ориентированные языки программирования»

Выполнил: студент ТКИ-341 Козлов А. Д.

Проверили: Доцент “УиЗИ” Логинова Л.Н.

Зав. лаборатории “УиЗИ” Антонов Д. А.

Москва 2024 г.

Цель работы: изучение логических команд и команд манипулирования битами, получение навыка работы с ними.

Постановка задачи:

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

   Результаты каждого пункта сохранить в памяти и распечатать.

Номер варианта	Число(HEX)
25	**58697EFF**

Код программы:

```
#include <iostream>
#include <sstream>
#include <bitset>

using namespace std;
const int DataSize = 8;

string IntToHex(int32_t n)
{
	stringstream ss;
	ss << hex << n;
	return ss.str();
}


int main()
{
	setlocale(LC_ALL, "ru");
	int32_t data[DataSize] = { NULL };

	int32_t num = 0x58697EFF;
	int8_t unit_counter = 0;
	int8_t zero_counter = 0;

	__asm {
		PUSHAD
		LEA ESI, num
		LEA EDI, data
		MOV ECX, 0
		CALL counter_with_ROL_proc
		CALL counter_with_TEST_proc
		CALL pair_counter_proc
		CALL reverse_lower_byte_proc

		JMP exit_mark

		counter_with_ROL_proc:
			PUSH ECX
			MOV EAX, DWord Ptr [ESI]
			MOV ECX, 32
			xor EBX, EBX
			xor EDX, EDX

			counter_with_ROL_proc_loop:
				ROL EAX, 1
				JC ROL_unit_bit_condition
				INC EDX
				JMP end_ROL_loop

				ROL_unit_bit_condition:
				INC EBX
				JMP end_ROL_loop

			end_ROL_loop:
			LOOP counter_with_ROL_proc_loop
			POP ECX
			MOV DWord Ptr [EDI + ECX * 4], EBX
			INC ECX
			MOV DWord Ptr [EDI + ECX * 4], EDX
			INC ECX
			RET
```

```
counter_with_TEST_proc:
	PUSH ECX
	MOV EAX, DWord Ptr [ESI]
	MOV ECX, 32
	xor EBX, EBX
	xor EDX, EDX

	counter_with_TEST_proc_loop:
		TEST EAX, 1
		JZ TEST_unit_bit_condition
		INC EBX
		JMP end_TEST_loop

		TEST_unit_bit_condition:
		INC EDX
		JMP end_TEST_loop

	end_TEST_loop:
	SHR EAX, 1
	LOOP counter_with_TEST_proc_loop
	POP ECX
	MOV DWord Ptr [EDI + ECX * 4], EBX
	INC ECX
	MOV DWord Ptr [EDI + ECX * 4], EDX
	INC ECX
	RET

pair_counter_proc:
	PUSH ECX
	MOV EAX, DWord Ptr [ESI]
	MOV ECX, 31
	MOV EBX, 0x3

	pair_counter_loop:
		PUSH EAX
		and EAX, EBX
		CMP EAX, EBX
		JZ unit_pair_condition
		POP EAX
		PUSH EAX
		not EAX
		and EAX, EBX
		CMP EAX, EBX
		JZ zero_pair_condition
		JMP end_pair_counter_loop

		unit_pair_condition:
		MOV DL, Byte Ptr [unit_counter]
		INC DL
		MOV Byte Ptr [unit_counter], DL
		JMP end_pair_counter_loop

		zero_pair_condition:
		MOV DL, Byte Ptr [zero_counter]
		INC DL
		MOV Byte Ptr [zero_counter], DL
		JMP end_pair_counter_loop

	end_pair_counter_loop:
	POP EAX
	SHL EBX, 1
	LOOP pair_counter_loop
	POP ECX
	MOVZX EDX, Byte Ptr [unit_counter]
	MOV DWord Ptr [EDI + ECX * 4], EDX
```

```
                INC ECX
                MOVZX EDX, Byte Ptr [zero_counter]
                MOV DWord Ptr [EDI + ECX * 4], EDX
                INC ECX
                RET


        reverse_lower_byte_proc:
                PUSH ECX
                MOV AL, Byte Ptr [ESI]
                MOV DH, 0x80
                MOV DL, 0x01
                MOV CL, 4

                reverse_loop:
                        CALL get_reverse_bits_by_mask_proc
                        CALL clear_mask_bits_proc
                        or AL, BL
                        SHR DH, 1
                        SHL DL, 1
                LOOP reverse_loop
                POP ECX
                MOV EDX, num
                MOV DWord Ptr [EDI + ECX * 4], EDX
                INC ECX
                MOV DL, AL
                MOV DWord Ptr [EDI + ECX * 4], EDX
                INC ECX
                RET

                get_reverse_bits_by_mask_proc:
                        MOV BH, AL
                        MOV BL, AL
                        and BH, DH
                        and BL, DL
                        CALL shift_mask_bits_value
                        or BL, BH
                        RET

                shift_mask_bits_value:
                        PUSH CX
                        SHL CL, 1
                        DEC CL
                        SHR BH, CL
                        SHL BL, CL
                        POP CX
                        RET

                clear_mask_bits_proc:
                        MOV BH, DH
                        or BH, DL
                        not BH
                        and AL, BH
                        RET

        exit_mark:
        POPAD
    }

    int i = 0;
    bitset<32> bin_num(num);
    cout << "Посчет кол-ва единиц и нулей %1 (команда сдвига ROL)" << endl;
    cout << "Число: \t" << num << " \thex: " << IntToHex(num) << " \tbin: " <<
bin_num << endl;
    cout << "Единиц: " << data[i++] << endl;
```

```
	cout << "Нулей: " << data[i++] << "\n" <<  endl;

	cout << "Посчет кол-ва единиц и нулей %2 (команда TEST)" << endl;
	cout << "Число: \t" << num << " \thex: " << IntToHex(num) << " \tbin: " <<
bin_num << endl;
	cout << "Единиц: " << data[i++] << endl;
	cout << "Нулей: " << data[i++] << "\n" << endl;

	cout << "Посчет кол-ва пар единиц и нулей" << endl;
	cout << "Число: \t" << num << " \thex: " << IntToHex(num) << " \tbin: " <<
bin_num << endl;
	cout << "Пар единиц: " << data[i++] << endl;
	cout << "Пар нулей: " << data[i++] << "\n" << endl;

	cout << "Отображение последнего байта числа" << endl;
	bitset<32> source_bin_num(data[i]);
	cout << "Исходное число: \t" << data[i] << " \thex: " << IntToHex(data[i]) <<
" \tbin: " << source_bin_num << endl;
	i++;
	bitset<32> target_bin_num(data[i]);
	cout << "Полученное число: \t" << data[i] << " \thex: " << IntToHex(data[i])
<< " \tbin: " << target_bin_num << endl;
}
```

Ход выполнения:

Заданное число:

$$\mathbf{58697EFF_{16}}$$

$$\mathbf{0101\ 1000\ 0110\ 1001\ 0111\ 1110\ 1111\ 1111_2}$$

$$\mathbf{1\ 483\ 308\ 799_{10}}$$

В регистре ECX храним индекс элемента в массиве элементов Data, а при добавлении элемента инкрементируем его.

1. Подсчет единиц и нулей с помощью команды *ROL*.

   Сдвигаем по кругу влево исходное число с помощью команды *ROL*, флаг CF меняется на 1, если старший бит был 1, иначе 0. В зависимости от условия инкрементируем счетчик нулей или единиц.

2. Подсчет единиц и нулей с помощью команды *TEST*.

   Аналогично первому варианту сдвигаем исходное слово. В данном варианте с помощью команды *SHR* в правую сторону и проверяем командой *TEST* младший бит числа. В зависимости от условия инкрементируем счетчик нулей или единиц.

3. Подсчет пар единиц и нулей в бинарном представлении числа.

   Подсчет пар единиц осуществляется логический умножением исходного числа на маску 11_2 и постепенным сдвигом на одну позицию влево самой маски.

   Подсчет нулей также происходит путем перемножения числа на маску 11_2, но перед этим число должно инвертироваться командой *not*.

   Одновременно идет проверка и на единичную пару, и на нулевую. Т.к. число 32-битное, а маска состоит из 2-х битов, значит всего будет 31 итерация.

4. Отображение битов младшего байта числа.

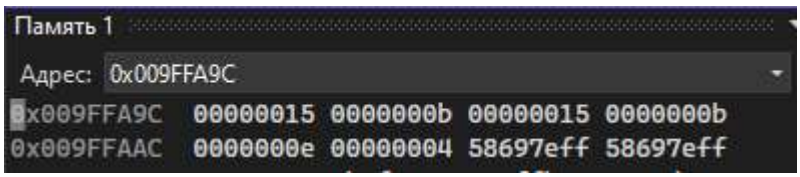
   Отображение битов осуществляется путем вычисления маски для каждой итерации и наложения этой маски на исходное число, для получения битов исходных битов, а затем зеркальное отображение этих битов. Затем в исходном числе обнуляются биты по текущей маске и производится логическое сложение отраженных битов и числа с обнуленными битами.

   Каждую итерацию маска вычисляется путем сдвига старшего бита числа вправо на одну позицию, а младшего влево на одну позицию. Тем самым мы постепенно проходим по всем парам числа и

меняем биты местами. Т.к. мы имеем дело с 8 битным числом, значит всего будет выполнено 4 итерации.

В результате работы программы в памяти будут последовательно лежать 8 элементов:

1. Первая пара элементов – число единиц и нулей в числе после подсчета командой *ROL*.
2. Вторая пара элементов – число единиц и нулей в числе после подсчета командой *TEST*.
3. Третья пара элементов – число пар единиц и нулей в числе
4. Четвертая пара – исходное число и число после отражения младшего байта числа.




```
Посчет кол-ва единиц и нулей №1 (команда сдвига ROL)
Число:  1483308799      hex: 58697eff   bin: 01011000011010010111111011111111
Единиц: 21
Нулей: 11

Посчет кол-ва единиц и нулей №2 (команда TEST)
Число:  1483308799      hex: 58697eff   bin: 01011000011010010111111011111111
Единиц: 21
Нулей: 11

Посчет кол-ва пар единиц и нулей
Число:  1483308799      hex: 58697eff   bin: 01011000011010010111111011111111
Пар единиц: 14
Пар нулей: 4

Отображение последнего байта числа
Исходное число:         1483308799      hex: 58697eff   bin: 01011000011010010111111011111111
Полученное число:       1483308799      hex: 58697eff   bin: 01011000011010010111111011111111
```

Как видим в результате первого и второго способа подсчета получаем одинаковое число единиц и нулей. Также при подсчете пар нулей и единиц получаем 14 пар единиц и 4 пары нулей. В заданном числе последний бит равен FF_{16}, что равно $1111\ 1111_2$, а значит при отражении числа получается исходное число.

Вывод:

1. В ходе выполнения лабораторной работы были изучены логические команды и команды манипулирования битами.
2. Были изучены команды битовых сдвигов такие, как SHR, ROL и другие.
3. Также была изучена группа команд LOOP, которые позволяют создавать циклы со счетчиком.
4. Был продемонстрирован подход в оперировании битами числа с помощью масок в связке с логическими командами.

Заключение

Выполнение лабораторной работы расширило знание команд языка ассемблера. Разными подходами научились работе с битами числа. Полученные навыки будут необходимы при точечной работе с битами числа, битами прав и настроек, а также может ускорить умножение чисел на 2 путем замены обычной операции умножения на битовый сдвиг числа.