|  |  |
| --- | --- |
|  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **«Московский технологический университет»**  **МИРЭА** | | |

Институт Информационных Технологий

Кафедра Корпоративных Информационных Систем

**ОТЧЕТ**

по Лабораторной Работе №4

на тему

«Программирование на языке ассемблера»

по дисциплине

«Технологии программирования»

Выполнил студент группы ИСБО-11-16 Шайхуллин С.В.

Принял Cтарший преподаватель Мирзоян Д.И.

Выполнено «\_\_\_» \_\_\_\_\_\_\_\_\_2017 г.

Зачтено «\_\_\_» \_\_\_\_\_\_\_\_\_ 2017г.

Москва, 2017

**Теоретическая часть**

Язык ассемблера — низкоуровневый язык программирования, представляющий собой человеко-понятный способ представления машинных команд. Язык ассемблера является исторически первым из появившихся языков — он возник сразу после непосредственного программирования в машинных командах. Язык ассемблера является абсолютно непереносимым — из-за низкоуровневости, он жестко привязан к системе команд одного процессора (семейства процессоров).

В ассемблере один оператор языка преобразуется в процессе ассемблирования в одну машинную команду.

Из-за непосредственной связи языка ассемблера и системы команд процессора, операторы языка не являются универсальными, и являются тестовыми записями машинных команд. Чаще всего это аббревиатура, обозначающая то или иное машинное действие.

Например, команда пересылки данных (MOVE)

mov eax, 22

В языке ассемблера сначала указывается команды, затем идут операнды в порядке ОПЕРАНД-ПРИЕМНИК, ОПЕРАНД-ИСТОЧНИК (синтаксис Intel). Существует обратный вариант записи операндов (синтаксис AT&T).

Язык ассемблера в настоящее время применяется достаточно редко — существующие языки высокого уровня и их компиляторы достаточно удобны и эффективны. Ассемблерный код применяется в 3 основных случаях:

1. Время выполнения данного участка сверхкритично, и программист применяет ручное написание кода с какими-то неочевидными низкоуровневыми оптимизациями
2. Используются команды и режимы, неизвестные компилятору, скажем MMX, SIMD-расширения
3. Используются системные регистры, возможности и режимы процессора (создание ОС, системного и диагностического ПО)

Для решения задач 1-й и 2-й областей обычно не пишут программу или модуль программы на ассемблере целиком — это неэффективно, с точки зрения трудозатрат (эффективнее написать вручную только самые критичные участки кода, а остальное доверить компилятору — современный среднестатистический компилятор знает ассемблер заметно лучше современного среднестатистического программиста). В таком случае используют технологию ассемблерных вставок — в программу на языке высокого уровня включаются фрагменты на языке низкого уровня (ассемблере). Например, вычисление суммы двух чисел будет выглядеть так:

int Add(int a, int b)

{

\_\_asm {

mov eax, a

mov ebx, b

add eax, ebx

}

}

В данном случае используется синтаксис Intel. Тот же фрагмент на синтаксисе AT&T выглядит так:

int Add(int a, int b)

{

\_\_asm {

movl A, %eax

movl B, %ebx

addl %ebx, %eax

}

}

(Данный пример не будет работать с компилятором MSVC. Но применим, например, для компилятора gcc.)

В ассемблерных вставках вы можете использовать любые инструкции, символьные имена переменных и функций, а также любые регистры, но значения регистров EBP, ESP, а также сегментных регистров должны остаться неизменными на момент выхода из ассемблерной вставки. Для сохранения значения регистров можно использовать пару команд push/pop.

Обратите внимание, что использование ассемблерных вставок запрещено в x64 коде, а также в любом управляемом. Поэтому, для реализации задания вам потребуется создать дополнительный проект неуправляемой библиотеки C++, в которой вы и будете реализовывать функции задания с помощью ассемблерных вставок. Также вам следует помнить о соглашениях именования функции в C++/C и о способах передачи значений параметров в тело функции (соглашениях вызова).

**Задание**

Создать приложение, осуществляющее решение индивидуального задания с использованием ассемблерных вставок.

**Пример работы**

**Заключение**

В данной лабораторной работе мы научились выполнять простые операции на языке ассемблера NASM, работать с прерываниями и системными вызовами, работать с регистрами, оперативной памятью и стеком потока

**Исходный код**

even.asm

global main

extern printf

section .data

fmt db '%d', 10, 0

arr dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 11, 11, 11, 12

len equ($ - arr) / 4

; Количество четных чисел в массиве

section .text

main:

xor eax, eax

mov ecx, 0

.for:

mov edx, [arr + 4 \* ecx]

inc ecx

test edx, 1

jz .even

jnz .compare

.even:

inc eax

.compare:

cmp ecx, len

jl .for

je .show

.show:

push eax

push fmt

call printf

mov eax, 1

xor ebx, ebx

int 0x80

join.asm

global main

extern printf

section .data

fmt db '%d', 10, 0

arr1 dd 1, 2, 3, 4, 5

len equ($ - arr1) / 4

arr2 dd 1, 2, 3, 4, 5

section .bss

arr3 resd len \* 4

; Поэлементно сложить два массива

section .text

main:

xor esi, esi

.for:

mov eax, [arr1 + 4 \* esi]

add eax, [arr2 + 4 \* esi]

mov [arr3 + 4 \* esi], eax

.show:

push eax

push fmt

call printf

.compare:

inc esi

cmp esi, len

jl .for

mov eax, 1

xor ebx, ebx

int 0x80

max.asm

global main

extern printf

section .data

fmt dw '%d', 10, 0

arr dd 1, 3, 5, 4, 2

len equ($ - arr) / 4

section .text

main:

xor eax, eax

mov ecx, len

mov esi, arr

.for:

mov edx, [esi]

add esi, 4

cmp eax, edx

jl .max

jg .compare

.max:

mov eax, edx

.compare:

dec ecx

cmp ecx, 0

je .show

jne .for

.show:

push eax

push fmt

call printf

mov eax, 1

xor ebx, ebx

int 0x80

sum.asm

global main

extern printf

section .data

fmt dw '%d', 10, 0

arr dd 1, 2, 3, 4, 5, 6

len equ ($ - arr) / 4

section .text

main:

mov esi, arr

xor eax, eax

mov ecx, len

.for:

add eax, [esi]

add esi, 4

loop .for

push eax

push fmt

call printf

mov eax, 1

xor ebx, ebx

int 0x80

Makefile

MAKEPART=MakePart

all: max even join sum

max:

make -f $(MAKEPART) PART='max'

even:

make -f $(MAKEPART) PART='even'

join:

make -f $(MAKEPART) PART='join'

sum:

make -f $(MAKEPART) PART='sum'

MakePart

export PART

SOURCE=src/$(PART)

BIN=bin/$(PART)

all: nasm ld rm

nasm:

nasm -f elf $(SOURCE).asm -o $(BIN).o

LINKER=-dynamic-linker /lib/ld-linux.so.2 -lc

ld:

ld $(BIN).o -o $(BIN) -e main -melf\_i386 $(LINKER)

rm:

rm $(BIN).o