###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«ВВЕДЕНИЕ В АРХИТЕКТУРУ x86/x86-64»

студента 2 курса, 18209 группы

**Большим Максима Антоновича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Власенко Андрей Юрьевич

Новосибирск 2019

1. **Цель**

1. Изучить аспекты работы с языком ассемблера;

2. Ознакомиться с архитектурой x86/x86-64;

3. Провести детальный анализ ассемблерного кода программы 1й лабораторной работы.

1. **Задание**

Изучить программную архитектуру x86/x86-64. Ознакомиться с набором регистров процессора, основными арифметико-логическими командами, командами арифметического сопроцессора. Изучить способы адресации памяти, работу со стеком, так же понять принцип вызова подпрограмм, передачу параметров в подпрограммы и возврат результатов. Для программы на языке С++ сгенерировать ассемблерные листинги для архитектуры x86 и архитектуры x86-64, после чего провести анализ ассемблерных листингов, сопоставляя команды языка С++ с машинными командами. После детального анализа процессорных инструкций продемонстрировать использование ключевых особенностей архитектур x86 и x86-64 на конкретных участках кода, после чего сравнить различия в программах для архитектуры x86 и архитектуры x86-64.

1. **Листинг программы**

**#include <iostream>**

**inline double power(unsigned long power)**

**{**

**if(power % 2 == 1)**

**return -1.0;**

**else return 1.0;**

**}**

**int main()**

**{**

**unsigned long N = 120000000000;**

**double Pi = 0;**

**for(unsigned long i = 0;i < N;i++)**

**Pi += power(i) \* (4.0L /**

**(2.0L \* static\_cast<long double>(i) + 1.0L));**

**printf("%.20f\n", Pi);**

**}**

1. **Ассемблерный листинг для архитектуры x86-64**

power(unsigned long):

pushq %rbp // save rbp

movq %rsp, %rbp // stack top to rbp

movq %rdi, -8(%rbp) // load power to stack

movq -8(%rbp), %rax // move power to rax

andl $1, %rax // % 2

testq %rax, %rax // check to zero

je .L2 // == 0

movsd .LC0(%rip), %xmm0 // load -1.0 to xmm0

jmp .L3 // return answer and load rbp to back

.L2:

movsd .LC1(%rip), %xmm0 // load 1.0 to xmm0

.L3:

popq %rbp

ret

.LC6:

.string "%.20d\n"

main:

pushq %rbp

movq %rsp, %rbp

subq $48, %rsp

movabsq $12000000000, %rax // load N to rax

movq %rax, -24(%rbp) // load N to rbp-24

pxor %xmm0, %xmm0 // Pi = 0

movsd %xmm0, -8(%rbp) // move Pi to rbp-8

movq $0, -16(%rbp) // load i to rbp-16

.L7:

movq -16(%rbp), %rax

cmpq -24(%rbp), %rax // compare N with i

jnb .L5 // N <= i

movq -16(%rbp), %rax // load i

movq %rax, %rdi

call power(unsigned long)// put i to power and call

movsd %xmm0, -40(%rbp) // put result to rbp-40

fldl -40(%rbp) // load +-1.0 to float sp stack st(0)

fildq -16(%rbp) // load i to float sp stack

cmpq $0, -16(%rbp) // compare i with 0

jns .L6 // i > 0

fldt .LC3(%rip)

faddp %st, %st(1)

.L6:

fld %st(0) // put i to sp stack

faddp %st, %st(1) // 2i

fld1 // load 1 to stack

faddp %st, %st(1) // 2i + 1

fldt .LC5(%rip) // load 4.0

fdivp %st, %st(1) // 4 / (2i + 1)

fmulp %st, %st(1) // +-1 \* 4/(2i + 1)

fldl -8(%rbp) // load Pi

faddp %st, %st(1) // Pi + (+-1 \* 4/(2i + 1))

fstpl -8(%rbp) // unload new Pi

addq $1, -16(%rbp)//i++

jmp .L7

.L5:

movq -8(%rbp), %rax

movq %rax, %xmm0 // Pi to xmm0

movl $.LC6, %edi // format to edi

movl $1, %eax

call printf

movl $0, %eax

leave

ret // return 0 and end program

\_\_static\_initialization\_and\_destruction\_0(int, int):

pushq %rbp

movq %rsp, %rbp

subq $16, %rsp

movl %edi, -4(%rbp)

movl %esi, -8(%rbp)

cmpl $1, -4(%rbp)

jne .L11

cmpl $65535, -8(%rbp)

jne .L11

movl $\_ZStL8\_\_ioinit, %edi

call std::ios\_base::Init::Init() [complete object constructor]

movl $\_\_dso\_handle, %edx

movl $\_ZStL8\_\_ioinit, %esi

movl $\_ZNSt8ios\_base4InitD1Ev, %edi

call \_\_cxa\_atexit

.L11:

nop

leave

ret

\_GLOBAL\_\_sub\_I\_main:

pushq %rbp

movq %rsp, %rbp

movl $65535, %esi

movl $1, %edi

call \_\_static\_initialization\_and\_destruction\_0(int, int)

popq %rbp

ret

.LC0:

.long 0

.long -1074790400

.LC1:

.long 0

.long 1072693248

.LC3:

.long 0

.long 2147483648

.long 16447

.long 0

.LC5:

.long 0

.long 2147483648

.long 16385

.long 0

1. **Ассемблерный листинг для архитектуры x86**

power(unsigned long):

pushl %ebp // save ebp

movl %esp, %ebp // esp to ebp

movl 8(%ebp), %eax // power to eax

andl $1, %eax // power % 2

testl %eax, %eax // power % 2 == 0

je .L2 // == 0

fld1

fchs //else load -1 to st(0)

jmp .L3

.L2:

fld1 //load 1 to st(0)

.L3:

popl %ebp //unload ebp

ret

.LC7:

.string "%.20d\n"

main:

// Copy esp+4 into ecx. The value at [esp] has the return address, so esp+4 is 'argc', or the start of the function's arguments.

leal 4(%esp), %ecx

// Round esp down (align esp down) to the nearest 16-byte boundary.

// This ensures that regardless of what esp was before, esp is now

// starting at an address that can store any register this processor has,

// from the one-byte registers all the way up to the 16-byte xmm registers.

andl $-16, %esp

// Since we copied esp+4 into ecx above, that means that [ecx] is 'argc',

// [ecx+4] is 'argv', and [ecx+8] is 'envp'. For whatever reason, the

// compiler decided to push a duplicate copy of 'argv' onto the function's

// new local frame.

pushl -4(%ecx)

pushl %ebp // save ebp

movl %esp, %ebp // esp to ebp

// Preserve 'ecx'. Ecx tells us what 'esp' was before we changed 'esp'

// in the 'and'-instruction above, so we'll need it later to restore

// 'esp' before we return.

pushl %ecx

subl $36, %esp // new 36 bytes

movl $-884901888, -24(%ebp) // put N

fldz // load zero at st(0) (Pi)

fstpl -16(%ebp) // load Pi to ebp-16

movl $0, -20(%ebp) // load i to ebp-20

.L6:

movl -20(%ebp), %eax // load i to eax

cmpl -24(%ebp), %eax // cmp N and i

jnb .L5 // N <= i

subl $12, %esp // "memory" for power

pushl -20(%ebp) // i to stack

call power(unsigned long) // call power with i

addl $16, %esp // free "memory" for power

movd -20(%ebp), %xmm0 // load i ti xmm0

movq %xmm0, -32(%ebp) // i to ebp-32

fildq -32(%ebp) // i to st(0)

fld %st(0) // i to st(1)

faddp %st, %st(1) // 2i

fld1 // load 1 to st(0)

faddp %st, %st(1) // 2i + 1

fldt .LC5 // load 4 to st(0)

fdivp %st, %st(1) // 4 / (2i + 1)

fmulp %st, %st(1) // +-1 \* 4 / (2i + 1)

fldl -16(%ebp) // load Pi

faddp %st, %st(1) // update Pi += +-1 \* 4 / (2i + 1)

fstpl -16(%ebp) // unload Pi back

addl $1, -20(%ebp) // i++

jmp .L6

.L5:

subl $4, %esp // esp -= 4

pushl -12(%ebp) //

pushl -16(%ebp) // put Pi

pushl $.LC6 // put format

call printf

addl $16, %esp // free memory from printf

movl $0, %eax // will return 0

movl -4(%ebp), %ecx // get saved ecx

leave

leal -4(%ecx), %esp // restore old value of esp

ret // return 0

\_\_static\_initialization\_and\_destruction\_0(int, int):

pushl %ebp

movl %esp, %ebp

subl $8, %esp

cmpl $1, 8(%ebp)

jne .L10

cmpl $65535, 12(%ebp)

jne .L10

subl $12, %esp

pushl $\_ZStL8\_\_ioinit

call std::ios\_base::Init::Init() [complete object constructor]

addl $16, %esp

subl $4, %esp

pushl $\_\_dso\_handle

pushl $\_ZStL8\_\_ioinit

pushl $\_ZNSt8ios\_base4InitD1Ev

call \_\_cxa\_atexit

addl $16, %esp

.L10:

nop

leave

ret

\_GLOBAL\_\_sub\_I\_main:

pushl %ebp

movl %esp, %ebp

subl $8, %esp

subl $8, %esp

pushl $65535

pushl $1

call \_\_static\_initialization\_and\_destruction\_0(int, int)

addl $16, %esp

leave

ret

.LC5:

.long 0

.long 2147483648

.long 16385

1. **Ассемблерный листинг для архитектуры x86-64 с оптимизацией O3**

.LC3:

.string "%.20f\n"

main:

subq $24, %rsp

flds .LC2(%rip)

movsd .LC0(%rip), %xmm1

movl $1, %eax

movabsq $120000000000, %rdx

movsd %xmm1, (%rsp)

fld1

fld %st(0)

.L5:

movq %rax, 8(%rsp)

fildq 8(%rsp)

fadd %st(0), %st

testb $1, %al

jne .L2

fadd %st(1), %st

addq $1, %rax

fdivr %st(3), %st

faddl (%rsp)

fstpl (%rsp)

cmpq %rdx, %rax

jne .L5

fstp %st(0)

fstp %st(0)

fstp %st(0)

.L3:

movsd (%rsp), %xmm0

movl $.LC3, %edi

movl $1, %eax

call printf

xorl %eax, %eax

addq $24, %rsp

ret

.L2:

fadd %st(2), %st

addq $1, %rax

fdivr %st(3), %st

fsubrl (%rsp)

fstpl (%rsp)

cmpq %rdx, %rax

jne .L5

fstp %st(0)

fstp %st(0)

fstp %st(0)

jmp .L3

\_GLOBAL\_\_sub\_I\_main:

subq $8, %rsp

movl $\_ZStL8\_\_ioinit, %edi

call std::ios\_base::Init::Init() [complete object constructor]

movl $\_\_dso\_handle, %edx

movl $\_ZStL8\_\_ioinit, %esi

movl $\_ZNSt8ios\_base4InitD1Ev, %edi

addq $8, %rsp

jmp \_\_cxa\_atexit

.LC0:

.long 0

.long 1074790400

.LC2:

.long 1082130432

1. **Ассемблерный листинг для архитектуры x86 с оптимизацией O3**

.LC2:

.string "%.20f\n"

main:

leal 4(%esp), %ecx

andl $-16, %esp

movl $1, %eax

pushl -4(%ecx)

pushl %ebp

movl %esp, %ebp

pushl %ecx

subl $20, %esp

flds .LC0

jmp .L5

.L17:

movd %eax, %xmm1

addl $1, %eax

movq %xmm1, -16(%ebp)

fildq -16(%ebp)

fadd %st(0), %st

fadds .LC1

fdivrs .LC0

faddp %st, %st(1)

fstpl -24(%ebp)

fldl -24(%ebp)

cmpl $-259084288, %eax

je .L16

.L5:

testb $1, %al

je .L17

movd %eax, %xmm0

addl $1, %eax

movq %xmm0, -16(%ebp)

fildq -16(%ebp)

fadd %st(0), %st

fadds .LC1

fdivrs .LC0

fsubrp %st, %st(1)

fstpl -24(%ebp)

fldl -24(%ebp)

cmpl $-259084288, %eax

jne .L5

.L16:

subl $12, %esp

fstpl (%esp)

pushl $.LC2

call printf

movl -4(%ebp), %ecx

addl $16, %esp

xorl %eax, %eax

leave

leal -4(%ecx), %esp

ret

\_GLOBAL\_\_sub\_I\_main:

subl $24, %esp

pushl $\_ZStL8\_\_ioinit

call std::ios\_base::Init::Init() [complete object constructor]

addl $12, %esp

pushl $\_\_dso\_handle

pushl $\_ZStL8\_\_ioinit

pushl $\_ZNSt8ios\_base4InitD1Ev

call \_\_cxa\_atexit

addl $28, %esp

ret

.LC0:

.long 1082130432

.LC1:

.long 1065353216

1. **Выводы**

По результатам проведённого анализа и сопоставления команд языка С++ и команд языка ассемблера было выяснено, что переменные, в зависимости от применённого уровня оптимизации при компиляции, хранились либо в стековом кадре, либо в регистрах общего назначения, преимущесвтенно регистрах rax, eax, xmm0. Основные различия архитектур заключались в размерах передаваемых данных: если же в x86 использовались 4х-байтные значения, то в x86-64 преимущественно 8-байтные значения. Особые отличия проявляются, когда используются типы на подобии unsigned long long, так как максимальный размер этого типа не поддерживается на x86. В остальном код для разных архитектур был одинаковым и мало где критически отличался. После анализа оптимизированного кода было выяснено, что в нём значительно уменьшено количество обращений в стэк, больше используются регистры общего назначения и некоторые команды выполняются заранее. Все вызовы inline функций убраны, а код функции помещён в основное тело программы.