实验2

班级：191211

姓名：孙鹤轩

学号：20211000156

实验内容：

1 熟悉基本汇编语言程序；

2 不同类型数据在计算机的编码、存储、转换，整型数据加减运算及其计算机底层实现，浮

点数据的表示与运算。

实验目标：

1 理解计算机中数据的表示、存储和运算，熟悉程序的机器级表示；

2 学习和掌握程序的调试方法，强化计算机编程实践能力；

3 掌握 C 语言中位操作语句的使用。

实验任务：

1 学习 MOOC 内容

https://www.icourse163.org/learn/NJU-1449521162

第三周 数据的存储与运算

第 1 讲 真值与机器数

第 2 讲 数据的宽度与存储

第 3 讲 数据类型的转换

第 4 讲 整数加减运算

第 5 讲 浮点数的表示和运算

第四周 程序的机器级表示

第 1 讲 传送指令

第 2 讲 加减运算指令

第 3 讲 整数乘法指令

2完成作业

2.1 C 语言程序如下，利用反汇编程序代码对运行结果进行解释说明。

#include “stdio.h”

void main()

{

int ai=100, bi=2147483648, ci=-100;

unsigned au=100, bu=2147483648, cu=-100;

printf(“ai =%d, bi=%d, ci=%d\n”, ai, bi, ci);

printf(“au =%u, bu=%u, cu=%u\n”, au, bu, cu);

}

2.1.1程序代码

反汇编代码如下：

0804848b <main>:

804848b: 8d 4c 24 04 lea 0x4(%esp),%ecx

804848f: 83 e4 f0 and $0xfffffff0,%esp

8048492: ff 71 fc pushl -0x4(%ecx)

8048495: 55 push %ebp

8048496: 89 e5 mov %esp,%ebp

8048498: 51 push %ecx

8048499: 83 ec 24 sub $0x24,%esp

804849c: c7 45 e0 64 00 00 00 movl $0x64,-0x20(%ebp)

80484a3: c7 45 e4 00 00 00 80 movl $0x80000000,-0x1c(%ebp)

80484aa: c7 45 e8 9c ff ff ff movl $0xffffff9c,-0x18(%ebp)

80484b1: c7 45 ec 64 00 00 00 movl $0x64,-0x14(%ebp)

80484b8: c7 45 f0 00 00 00 80 movl $0x80000000,-0x10(%ebp)

80484bf: c7 45 f4 9c ff ff ff movl $0xffffff9c,-0xc(%ebp)

80484c6: ff 75 e8 pushl -0x18(%ebp)

80484c9: ff 75 e4 pushl -0x1c(%ebp)

80484cc: ff 75 e0 pushl -0x20(%ebp)

80484cf: 68 9c 85 04 08 push $0x804859c

80484d4: e8 77 fe ff ff call 8048350 <printf@plt>

80484d9: 83 c4 10 add $0x10,%esp

80484dc: ff 75 f4 pushl -0xc(%ebp)

80484df: ff 75 f0 pushl -0x10(%ebp)

80484e2: ff 75 ec pushl -0x14(%ebp)

80484e5: 68 b2 85 04 08 push $0x80485b2

80484ea: e8 61 fe ff ff call 8048350 <printf@plt>

80484ef: 83 c4 10 add $0x10,%esp

80484f2: 90 nop

80484f3: 8b 4d fc mov -0x4(%ebp),%ecx

80484f6: c9 leave

80484f7: 8d 61 fc lea -0x4(%ecx),%esp

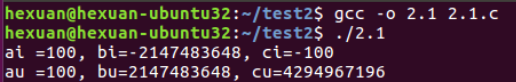
80484fa: c3 ret

80484fb: 66 90 xchg %ax,%ax

80484fd: 66 90 xchg %ax,%ax

80484ff: 90 nop  
2.1.2实验结果记录

程序的运行结果如下：



2.1.3结果分析与讨论

可以看到bi与cu的结果出现错误，分析如下：

首先我们从汇编代码中找出局部变量的存储位置和其赋值的机器数：

|  |  |  |
| --- | --- | --- |
| 变量名 | 存储位置 | 机器数 |
| ai | -0x20(%ebp) | 0x64 |
| bi | -0x1c(%ebp) | 0x80000000 |
| ci | -0x18(%ebp) | 0xffffff9c |
| au | -0x14(%ebp) | 0x64 |
| bu | -0x10(%ebp) | 0x80000000 |
| cu | -0xc(%ebp) | $0xffffff9c |

1为什么bi从正数变成了负数？

因为bi是int类型的数据，它的机器数是十六进制的0x80000000，程序中要求把bi输出为带符号中整数时，处理程序就把0x80000000当作补码数转换成真值，最高符号位由于为1，所以转换后的数是负数。

2无符号整数cu,赋值一个负的数据后，cu输出的结果为什么会是这个值?

cu是无符号整数，cu的机器数是十六进制的0xffffff9c，当程序中要求cu按照无符号整数输出时，处理程序就把0xffffff9c每一位都当作数值位来处理，cu输出的值为429496719。

ai和ci他们的值一个是100一个-100，从真值的角度讲他们只差一个符号位，但他们在计算机内部的机器数差异却很大，ai的机器数是0x00000064，ci的机器数是0xffffff9c，这是因为补码对正数和复数的编码值的差异。

cu的机器数是0xffffff9c，cu输出的真值是0xffffff9c的二进制值

2.2 C 语言程序如下，代码运行过程中各变量存储的机器数分别是什么？i1 和 i2 的值是否相同？f1 和 f2 的值是否相同？利用反汇编程序代码对结果进行解释说明。

#include “stdio.h”

int main()

{

int i1=0x7fffffff, i2, itemp;

float f1=0x987654321, f2, ftemp;

ftemp=i1;

i2=ftemp;

itemp=f1;

itemp=f1;

f2=itemp;

printf(“i1=%d, i2=%d, f1=%f, f2=%f\n”, i1, i2, f1, f2);

}

2.2.1程序代码

反汇编代码如下：

00000000 <main>:

#include <stdio.h>

int main()

{

0: 8d 4c 24 04 lea 0x4(%esp),%ecx

4: 83 e4 f0 and $0xfffffff0,%esp

7: ff 71 fc pushl -0x4(%ecx)

a: 55 push %ebp

b: 89 e5 mov %esp,%ebp

d: 51 push %ecx

e: 83 ec 34 sub $0x34,%esp

int i1=0x7fffffff, i2, itemp;

11: c7 45 e0 ff ff ff 7f movl $0x7fffffff,-0x20(%ebp)

float f1=0x987654321, f2, ftemp;

18: d9 05 1c 00 00 00 flds 0x1c

1e: d9 5d e4 fstps -0x1c(%ebp)

ftemp=i1;

21: db 45 e0 fildl -0x20(%ebp)

24: d9 5d e8 fstps -0x18(%ebp)

i2=ftemp;

27: d9 45 e8 flds -0x18(%ebp)

2a: d9 7d d6 fnstcw -0x2a(%ebp)

2d: 0f b7 45 d6 movzwl -0x2a(%ebp),%eax

31: b4 0c mov $0xc,%ah

33: 66 89 45 d4 mov %ax,-0x2c(%ebp)

37: d9 6d d4 fldcw -0x2c(%ebp)

3a: db 5d ec fistpl -0x14(%ebp)

3d: d9 6d d6 fldcw -0x2a(%ebp)

itemp=f1;

40: d9 45 e4 flds -0x1c(%ebp)

43: d9 6d d4 fldcw -0x2c(%ebp)

46: db 5d f0 fistpl -0x10(%ebp)

49: d9 6d d6 fldcw -0x2a(%ebp)

itemp=f1;

4c: d9 45 e4 flds -0x1c(%ebp)

4f: d9 6d d4 fldcw -0x2c(%ebp)

52: db 5d f0 fistpl -0x10(%ebp)

55: d9 6d d6 fldcw -0x2a(%ebp)

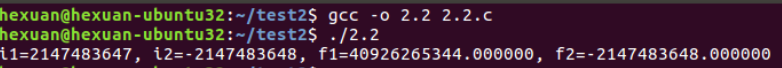
f2=itemp;

58: db 45 f0 fildl -0x10(%ebp)

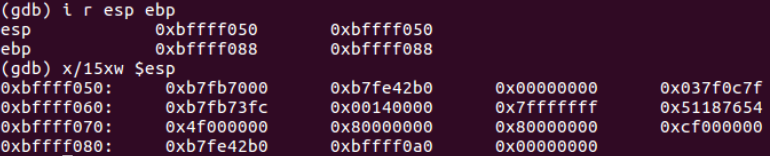
5b: d9 5d f4 fstps -0xc(%ebp)

printf("i1=%d, i2=%d, f1=%f, f2=%f\n", i1, i2, f1, f2);

2.2.2程序运行结果



设置断点在调用prinf函数前得到的栈帧结果如下：



2.2.3结果分析与讨论

|  |  |
| --- | --- |
| 变量名 | 机器数 |
| i1 | 0x7fffffff |
| i2 | 0x80000000 |
| itemp | 0x80000000 |
| f1 | 0x51187654 |
| f2 | 0x80000000 |
| ftemp | 0xcf000000 |

i1和i2的值不相同，因为在将i1从整数转换为浮点数时进行了舍入处理，导致最终的浮点数表示与原始整数有差异。i2的值是根据舍入后的浮点数的真值表示而来，而真值为负数，与i1的真值差异较大。

f1和f2的结果也不相同，因为f1的初始值超过了浮点数的精度范围，在转换为浮点数时进行了舍入操作。然后，将f1转换为整数时发生溢出，因为其二进制表示超过了整数类型的表示范围。这导致f2与f1的值不同，根本原因是f1超出了整数的表示范围，在转换为整数时发生了溢出。

2.3 C 语言程序如下，结合反汇编程序代码对运行过程中寄存器 eax，ebx，ecx 中的值进行

解释说明。

#include “stdio.h”

void main()

{

int p[2]={0x12345678,0x11223344};

asm

(

“lea -0x14(%ebp),%eax\n\t”

“mov -0x14(%ebp),%ebx\n\t”

“mov $1,%ecx\n\t”

“lea -0x14(%ebp,%ecx,4),%eax\n\t”

“mov -0x14(%ebp,%ecx,4),%ebx\n\t”

);

printf(“understand mov and lea\n”);

}

2.3.1源代码说明

部分反汇编代码：

00000000 <main>:

#include <stdio.h>

void main()

{

0: 8d 4c 24 04 lea 0x4(%esp),%ecx

4: 83 e4 f0 and $0xfffffff0,%esp

7: ff 71 fc pushl -0x4(%ecx)

a: 55 push %ebp

b: 89 e5 mov %esp,%ebp

d: 51 push %ecx

e: 83 ec 14 sub $0x14,%esp

11: 65 a1 14 00 00 00 mov %gs:0x14,%eax

17: 89 45 f4 mov %eax,-0xc(%ebp)

1a: 31 c0 xor %eax,%eax

int p[2]={0x12345678,0x11223344};

1c: c7 45 ec 78 56 34 12 movl $0x12345678,-0x14(%ebp)

23: c7 45 f0 44 33 22 11 movl $0x11223344,-0x10(%ebp)

asm

2a: 8d 45 ec lea -0x14(%ebp),%eax

2d: 8b 5d ec mov -0x14(%ebp),%ebx

30: b9 01 00 00 00 mov $0x1,%ecx

35: 8d 44 8d ec lea -0x14(%ebp,%ecx,4),%eax

39: 8b 5c 8d ec mov -0x14(%ebp,%ecx,4),%ebx

"mov -0x14(%ebp),%ebx\n\t"

"mov $1,%ecx\n\t"

"lea -0x14(%ebp,%ecx,4),%eax\n\t"

"mov -0x14(%ebp,%ecx,4),%ebx\n\t"

);

首先程序创建了一个长度为2的int数组，然后在程序中嵌入汇编语句，汇编语句含义如下：

ebp-14是p0的地址

"lea -0x14(%ebp),%eax\n\t" //将p0的地址赋值给eax

"mov -0x14(%ebp),%ebx\n\t" //将p0地址的值赋值给ebx

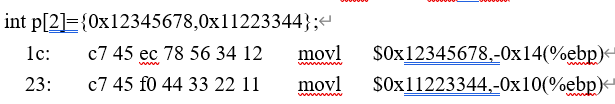
"mov $1,%ecx\n\t" //ecx相当于P[i]的下标i

ebp-14 + ecx\*4是p1的地址, int占4个字节

"lea -0x14(%ebp,%ecx,4),%eax\n\t" //p1的地址

"mov -0x14(%ebp,%ecx,4),%ebx\n\t" //p1地址上存储的数据

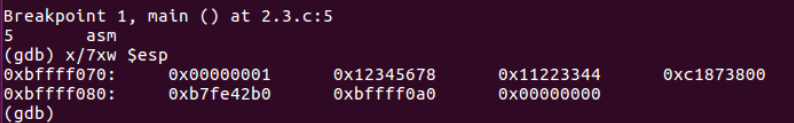
2.3.2实验结果



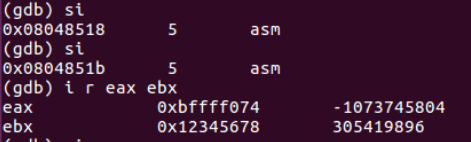
从汇编代码中可以看出p0的地址为-0x14(%ebp)，p1的地址为-0x10(%ebp)

通过gdb进行调试：

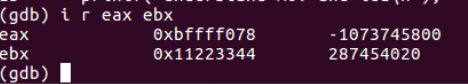
执行汇编代码前：



执行前两条汇编代码后



执行所有汇编指令后



2.3.3结果分析讨论

执行完前两个汇编语句后，lea指令将p0的地址传送给了eax寄存器，而mov指令将p0 的数值赋值给了ebx寄存器。

然后根据首地址和长度计算出p1的地址，再用同样的方法处理p1，得到的结果依然是eax接收p1的地址，ebx接收p1的值.

所以得出结论：lea指令实现的是地址传送，mov实现的是数据传送。

2.4 只用运算符～和|来实现位的与操作函数：

int bitAnd(int x, int y)

例如：bitAnd(6, 5) = 4

2.4.1程序代码与注释

# include<stdio.h>

int bitAnd(int x, int y){

return ~ (~x | ~y);// 将两个操作数先分别取反，然后做或操作，结果再取反

}

void main(){

int ans = bitAnd(6,5);

printf("%d\n",ans);

}

2.4.2实验结果



2.5 只用运算符! ~ & ^ | + << >>实现比较 x 和 y 的大小的函数：

int isLessOrEqual(int x, int y)

例如：isLessOrEqual(4, 5) = 1

2.5.1程序代码与注释

# include<stdio.h>

int isLessOrEqual(int a,int b) {

int diff = a ^ b;

if (!diff) return 1; //a = b

/\*\*001xxxxx -> 00100000，只需要找出第一个从最高位开始找出第一个不同的Bit，这一位是1的数是较大者\*\*/

diff |= diff >> 1;

diff |= diff >> 2;

diff |= diff >> 4;

diff |= diff >> 8;

diff |= diff >> 16;

diff ^= diff >> 1;

if (!(a & diff))

return 1; //a < b

else

return 0; //a > b

}

void main(){

//int a=1,b=5;

int ans = isLessOrEqual(4,5);

printf("%d\n",ans);

}

2.5.2实验结果

