

一、实验目的

- 1、熟悉 linux 系统的常用命令；
- 2、掌握 gcc 编译器的使用方法；
- 3、掌握 gdb 的调试工具使用；
- 4、掌握 objdump 反汇编工具使用；
- 5、理解反汇编程序（对照源程序与 objdump 生成的汇编程序）。

二、实验环境+-

列举所使用的软件工具


- 1、 Windows PowerShell
- 2、 Compiler Explorer

三、实验内容

现有两个 int 型数组 $a[i]=i-50$, $b[i]=i+y$, 其中 y 取自于学生本人学号 2022211x*y 的个位。登录 bupt1 服务器, 在 linux 环境下使用 vi 编辑器编写 C 语言源程序, 完成数组 a+b 的功能, 规定数组长度为 100, 函数名为 madd(), 数组 a, b 均定义在函数内, 采用 gcc 编译该程序(使用 **-g -fno-pie -fno-stack-protector** 选项),

- 1、使用 objdump 工具生成汇编程序, 找到 madd 函数的汇编程序, 给出截图;
- 2、用 gdb 进行调试, 练习下列 gdb 命令, 给出截图:
gdb、file、kill、quit、break、delete、clear、info break、run、continue、nexti、stepi、disassemble、list、print、x、info reg、watch
- 3、找到 $a[i]+b[i]$ 对应的汇编指令, 指出 $a[i]$ 和 $b[i]$ 位于哪个寄存器中, 给出截图;
- 4、使用单步指令及 gdb 相关命令, 显示 $a[xy]+b[xy]$ 对应的汇编指令执行前后操作数寄存器十进制和十六进制的值, 其中 x, y 取自于学生本人学号 2022211x*y 的百位和个位。

学号 2022211999, $a[99]+b[99]$ 单步执行前后的参考截图如下 (实际命令未显示出):

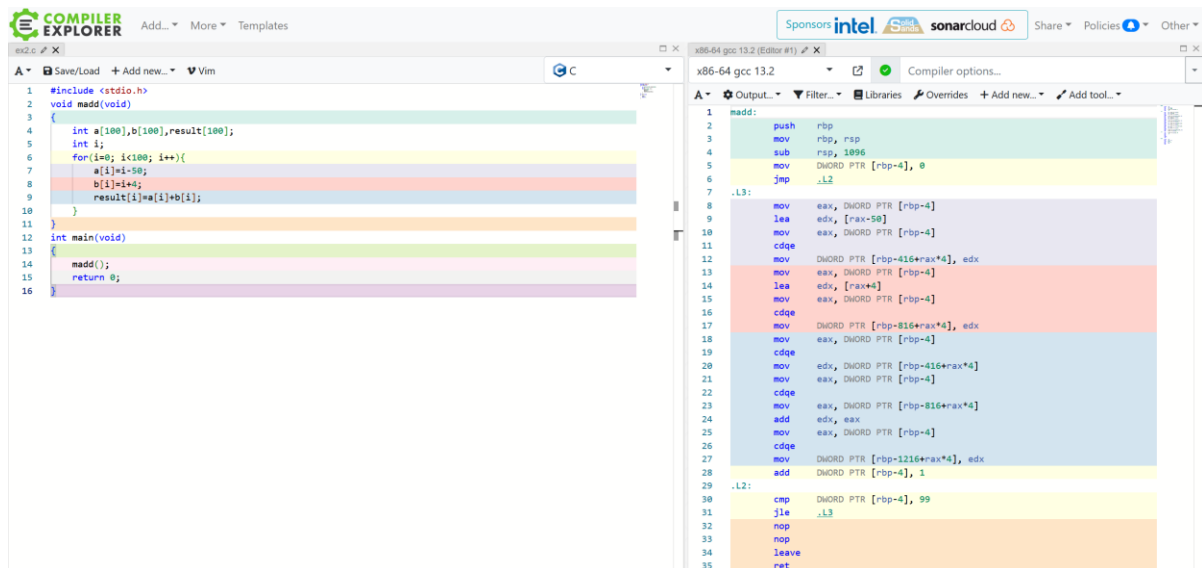


```
(gdb) 显示命令
$3 = 49
(gdb) 显示命令
$4 = 108
(gdb) 单步执行命令
0x00000000004005d8 in madd ()
(gdb) 显示命令
$5 = 157
(gdb) 显示命令
$6 = 108
```

四、实验步骤及实验分析

需要给出操作步骤、运行截图、分析过程的内容

1. 使用 vi 编辑器编写 c 程序 exp1.c, 保存并退出



Compiler Explorer 和 objdump 工具生成的反汇编目标文件一致（操作数是反的）

3. 用 gdb 进行调试

```

2022211414@bupt1:~$ gdb expl
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
(gdb) run
Starting program: /students/2022211414/expl
[Inferior 1 (process 1788057) exited normally]
(gdb) file expl
Load new symbol table from "expl"? (y or n) y
Reading symbols from expl...
(gdb) kill
The program is not being run.
(gdb) quit
2022211414@bupt1:~$ |
  
```

```

(gdb) break madd
Breakpoint 1 at 0x1155: file expl.c, line 6.
(gdb) delete 1
(gdb) break expl.c:6
Breakpoint 2 at 0x1155: file expl.c, line 6.
(gdb) clear expl.c:6

(gdb) info break
Deleted breakpoint 2 No breakpoints or watchpoints.
  
```

```

(gdb) break madd
Breakpoint 1 at 0x1155: file expl.c, line 6.
(gdb) run
Starting program: /students/2022211414/expl

Breakpoint 1, madd () at expl.c:6
6         for(i=0; i<100; i++){
(gdb) continue
Continuing.
[Inferior 1 (process 1971059) exited normally]
  
```

```

(gdb) list 1,16
1  #include <stdio.h>
2  void madd(void)
3  {
4      int a[100], b[100], result[100];
5      int i;
6      for(i=0; i<100; i++){
7          a[i]=i-50;
8          b[i]=i+4;
9          result[i]=a[i]+b[i];
10     }
11 }
12 int main(void)
13 {
14     madd();
15     return 0;
16 }
  
```

```

(gdb) nexti
0x0000555555551b0      6      for(i=0; i<100; i++){
(gdb) nexti
7      a[i]=i-50;
(gdb) info reg
rax      0x55555555551b5      93824992235957
rbx      0x55555555551d0      93824992235984
rcx      0x55555555551d0      93824992235984
rdx      0x7fffffffefb8      140737488350200
rsi      0x7fffffffefbe8      140737488350184
rdi      0x1                  1
rbp      0x7fffffffefae0      0x7fffffffefae0
rsp      0x7fffffffef698      0x7fffffffef698
r8       0x0                  0
r9       0x7ffff7fe0d60      140737354009952
r10      0xf                  15
r11      0x2                  2
r12      0x5555555555040      93824992235584
r13      0x7fffffffefbe0      140737488350176
r14      0x0                  0
r15      0x0                  0
rip      0x555555555515e      0x555555555515e <madd+20>
eflags   0x293                [ CF AF SF IF ]
cs       0x33                 51
ss       0x2b                 43
ds       0x0                  0
es       0x0                  0
fs       0x0                  0
gs       0x0                  0

```

```

(gdb) stepi
0x00005555555519a      9      result[i]=a[i]+b[i];
(gdb) p $edx
$6 = -50
(gdb) p $eax
$7 = 4
(gdb) stepi
0x00005555555519c      9      result[i]=a[i]+b[i];
(gdb) stepi
0x00005555555519f      9      result[i]=a[i]+b[i];
(gdb) stepi
0x0000555555551a1      9      result[i]=a[i]+b[i];
(gdb) stepi
6      for(i=0; i<100; i++){
(gdb) p $edx
$8 = -46
(gdb) x/d &result[0]
0x7fffffffef620: -46
(gdb) watch a[1]
Hardware watchpoint 2: a[1]
(gdb) continue
Continuing.

Hardware watchpoint 2: a[1]

Old value = 32767
New value = -49
madd () at expl.c:8
8      b[i]=i+4;
(gdb)

```

```
(gdb) disassemble madd
Dump of assembler code for function madd:
0x00005555555514a <+0>:    push    %rbp
0x00005555555514b <+1>:    mov     %rsp,%rbp
0x00005555555514e <+4>:    sub     $0x448,%rsp
=> 0x000055555555155 <+11>:   movl    $0x0,-0x4(%rbp)
0x00005555555515c <+18>:   jmp     0x555555551ac <madd+98>
0x00005555555515e <+20>:   mov     -0x4(%rbp),%eax
0x000055555555161 <+23>:   lea     -0x32(%rax),%edx
0x000055555555164 <+26>:   mov     -0x4(%rbp),%eax
0x000055555555167 <+29>:   cltq
0x000055555555169 <+31>:   mov     %edx,-0x1a0(%rbp,%rax,4)
0x000055555555170 <+38>:   mov     -0x4(%rbp),%eax
0x000055555555173 <+41>:   lea     0x4(%rax),%edx
0x000055555555176 <+44>:   mov     -0x4(%rbp),%eax
0x000055555555179 <+47>:   cltq
0x00005555555517b <+49>:   mov     %edx,-0x330(%rbp,%rax,4)
0x000055555555182 <+56>:   mov     -0x4(%rbp),%eax
0x000055555555185 <+59>:   cltq
0x000055555555187 <+61>:   mov     -0x1a0(%rbp,%rax,4),%edx
0x00005555555518e <+68>:   mov     -0x4(%rbp),%eax
0x000055555555191 <+71>:   cltq
0x000055555555193 <+73>:   mov     -0x330(%rbp,%rax,4),%eax
0x00005555555519a <+80>:   add     %eax,%edx
0x00005555555519c <+82>:   mov     -0x4(%rbp),%eax
0x00005555555519f <+85>:   cltq
0x0000555555551a1 <+87>:   mov     %edx,-0x4c0(%rbp,%rax,4)
0x0000555555551a8 <+94>:   addl    $0x1,-0x4(%rbp)
0x0000555555551ac <+98>:   cmpl    $0x63,-0x4(%rbp)
0x0000555555551b0 <+102>:  jle     0x5555555515e <madd+20>
0x0000555555551b2 <+104>:  nop
0x0000555555551b3 <+105>:  leaveq
0x0000555555551b4 <+106>:  retq
End of assembler dump.
(gdb) |
```

4. 在 vi 中查看 expl.s

a[i]位于%edx 中, b[i]位于%eax 中, a[i]+b[i]对应的汇编指令: addl %eax,%edx
运算结果 result[i]位于%edx 中

```
.L3:
    movl    -1220(%rbp), %eax
    leal    -50(%rax), %edx
    movl    -1220(%rbp), %eax
    cltq
    movl    %edx, -1216(%rbp,%rax,4)
    movl    -1220(%rbp), %eax
    leal    4(%rax), %edx
    movl    -1220(%rbp), %eax
    cltq
    movl    %edx, -816(%rbp,%rax,4)
    movl    -1220(%rbp), %eax
    cltq
    movl    -1216(%rbp,%rax,4), %edx → a[i]
    movl    -1220(%rbp), %eax
    cltq
    movl    -816(%rbp,%rax,4), %eax → b[i]
    addl    %eax, %edx
    movl    -1220(%rbp), %eax
    cltq
    movl    %edx, -416(%rbp,%rax,4)
    addl    $1, -1220(%rbp)
```

具体移动过程: 开始 a[i]和 b[i]赋值时都是运算完放在%edx, 然后移动到一处地址, 在做 a[i]+b[i]运算前再从两个地址中分别移动到%edx 和%eax

5. 显示 a[44]+b[44]对应的汇编指令执行前后操作数寄存器十进制和十六进制的值

进入函数, 监视 b[44]变化以移动到相加运算入口, 对照汇编程序单步调试至 addl 并打印寄存器内容

```
(gdb) watch b[44]
Hardware watchpoint 2: b[44]
(gdb) continue
Continuing.

Hardware watchpoint 2: b[44]

Old value = 0
New value = 48
madd () at expl.c:9
9      result[i]=a[i]+b[i];
```

a[44]+b[44]单步执行前后:

```
(gdb) p/d $edx
$1 = -6
(gdb) p/d $eax
$2 = 48
(gdb) p/x $edx
$3 = 0xfffffffffa
(gdb) p/x $eax
$4 = 0x30
(gdb) stepi
0x00005555555519c      9      result[i]=a[i]+b[i];
(gdb) p/d $edx
$5 = 42
(gdb) p/d $eax
$6 = 48
(gdb) p/x $edx
$7 = 0x2a
(gdb) p/x $eax
$8 = 0x30
```

执行前

执行后

五、总结体会

总结心得（包括实验过程中遇到的问题、如何解决的、意见和建议等）

我设置断点单步调试时，打印寄存器的内容发现 a[i]，b[i]和 result[i]都在%edx，i 在%eax，用 Compiler Explorer 测试了一下和 objdump 工具生成的结果一致，排除了编译选项的问题。但是这两种“分块”（显示源代码和汇编代码之间的对应关系）被我误解成了 a[i]和 b[i]在同一个寄存器。

后来仔细观察汇编发现，开始 i 在%eax，a[i]和 b[i]赋值时都是运算完放在%edx，然后移动到一处地址，在做 a[i]+b[i]运算前再从两个地址中分别移动到%edx 和%eax，运算后又把 i 放回%eax。由于我调试时总是看赋值后寄存器中的值所以会看到都在同一个寄存器的情况。

x 指令查看间接寻址的值时，比如写-816(%rbp,%rax,4)会报错，需要拆开写成-816+\$rbp+\$rax*4

以下截图为查看 b[0]:

```
(gdb) p $edx
$5 = 4
(gdb) x/d -816+$rbp+$rax*4
0x7fffffff7b0: 4
```

实验中我确实有不少疑惑，在不断翻书查资料以及与他人交流的过程中，我应用了课内所学，能对大部分问题做出合理的解释，从而对本门课程有了进一步的体会。通过本次实验，我熟悉了 linux 系统的常用命令，掌握了 gcc 编译器的使用方法，gdb 的调试工具使用，objdump 反汇编工具使用，对反汇编程序有更深刻的理解。这些技能对于开发和维护程序非常重要，在不断练习和深入了解这些概念和工具将有助于提高我的编程和调试技能。