一、 实验目的

1). 初步认识和掌握 MIPS 汇编语言程序设计的基本方法；

2). 熟悉 QEMU 模拟器和 GNU/Linux 下 x86-MIPS 交叉编译环境的使用。

二、 实验内容

1. 环境配置：

(1) 在虚拟机上安装交叉编译工具，包括：

- mips-linux-gnu-as：MIPS 交叉汇编器 op
- mips-linux-gnu-ld：MIPS 交叉链接器
- mips-linux-gnu-objdump：MIPS 交叉反汇编器
- mips-linux-gnu-readelf：MIPS ELF 文件查看器

安装完成后，可以直接调用上述工具进行面向 MIPS 架构的编译汇编和链接过程；

(2) 安装模拟运行环境：

在 X86 机器上，我们只能采用模拟器（如 QEMU，Bochs）运行 MIPS 程序，而不能使用虚拟机。虚拟机和模拟器原理不同，前者通过虚拟化硬件运行程序，只能运行和宿主机同架构系统，后者则采用软件模拟目标体系硬件。QEMU 是一款开源的、广为使用的跨平台模拟器，可以模拟出 x86，x86_64，MIPS，ARM 等等诸多指令集体系架构。QEMU 有用户模式和系统模式两种运行模式，前者是提供了一个类似于仅有 Linux 内核的运行环境，后者则需要自行指定启动系统所需的内核文件和 init 文件等，本实验中采取用户模式即可。

(3) 安装跨体系调试工具 gdb-multiarch 用于程序调试:

可直接在 linux 系统中输入

```
sudo apt install gdb-multiarch
```

指令进行安装

2. 编写 MIPS 程序并进行交叉编译，实现以下功能：

【排序】从键盘输入 10 个无符号字数或从内存中读取 10 个无符号字数并从大到小进行排序，排序结果在屏幕上显示出来。
例如对于输入

```
4 1 3 1 6 5 17 9 8 6
```

其输出为：

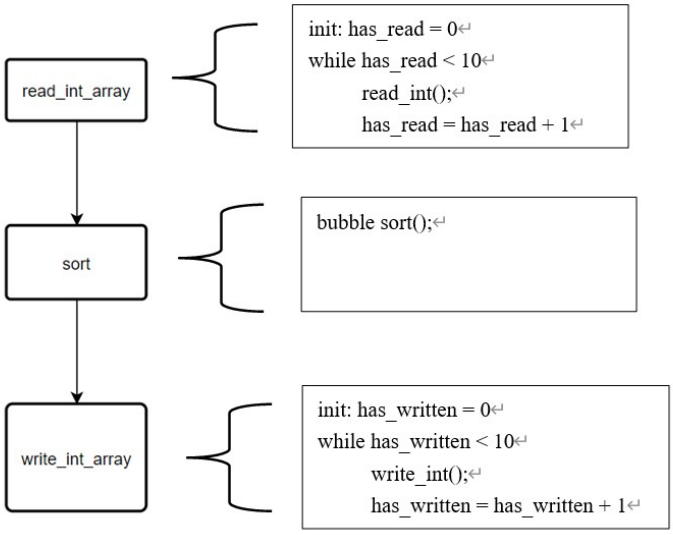
```
1 1 3 4 5 6 6 8 9 17
```

三、 实验器材

VMware 虚拟机器软件一套，已经安装好 GNU/Linux、QEMU 及 x86-MIPS 交叉编译环境的虚拟计算机一套。

四、 实验分析与设计

1. 程序流程：



2. 分析：

(1) 基于 QEMU 模拟环境运行的 MIPS 程序，是拥有一定的运行时环境的，即可以使用 Linux 内核所提供的系统调用，根据 MIPS 汇编语言知识可知，当 MIPS 程序直接运行于硬件上（裸机运行）时，指令 syscall 其实是使用微处理器提供的系统调用（处理器调用），然而在 QEMU 中由于增加了操作系统内核层面，syscall 将不能再直接使用处理器调用，而必须使用 Linux 内核所提供的系统调用（内核调用），以下是可用的部分内核调用：

```
//Defined in asm/unistd.h
#define __NR_Linux      4000
#define __NR_syscall      (__NR_Linux + 0)
#define __NR_exit     (__NR_Linux + 1)
#define __NR_fork     (__NR_Linux + 2)
#define __NR_read     (__NR_Linux + 3)
#define __NR_write      (__NR_Linux + 4)
#define __NR_open     (__NR_Linux + 5)
#define __NR_close      (__NR_Linux + 6)
```

由上可知，linux 的内核中仅有对于字符串操作的系统调用，而并不含对整数操作的系统调用，如 read_int，write_int 等，需要利用内核中的

```
extern ssize_t write (int __fd, const void *__buf, size_t __n) __wur;
extern ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur;
```

函数额外实现类似 atoi 和 itoa 的功能。

(2) 代码写好后，将在 x86 环境下编译上述代码并生成 MIPS 可执行文件，随后使用 QEMU 来运行。本实验中代码不需要预编译和预编译两个步骤，直接从汇编步骤开始；在汇编为可重定位目标文件时，使用 -g 指示加入符号信息以便于调试；链接生成可执行文件时，需要指定了程序入口点为 main 过程。最后使用 QEMU 所提供的 MIPS 用户模式运行交叉编译好的程序。

3. 设计：

(1) 自定义 read_int 函数部分代码：

```
ignore_space:

li      $v0,    4003
li      $a0,    0
la      $a1,    temp_char
li      $a2,    1
syscall
lb      $t0,    temp_char               #t0 stores temp
li      $t1,    ' '
beq     $t0,    $t1,    ignore_space    #go next inst if not ' ' else go to head
li      $t1,    '\n'
beq     $t0,    $t1,    ignore_space    #go next inst if not '\n' else go to head

```

功能：丢弃整数前的所有空格符与换行符
读入整数时，需要先丢弃输入中的‘ ’与‘\n’，实现方法为判断读入的字符是否为空格符或换行符，若是则丢弃，并循环执行 ignore_space

```
li      $t5,    0                       #t5 temporarily stores the return value
li      $t2,    10                      #base
li      $t3,    '0'                     #char to num

keep_reading_num:

li      $t1,    '0'
blt     $t0,    $t1,    return_back     #less than 0, not a num
li      $t1,    '9'
bgt     $t0,    $t1,    return_back     #greater than 9, not a num

mul     $t5,    $t5,    $t2             #t5 *= 10
sub     $t4,    $t0,    $t3             #t4 stores temp - '0'
add     $t5,    $t5,    $t4             #t5 = t5 + temp

```

```
16                  li      $v0,    4003
17                  li      $a0,    0
18                  la      $a1,    temp_char
19                  li      $a2,    1
20                  syscall
21
```

功能：对读入的字符串进行操作，计算出整数的值
读入整数时，设计思路为将逐个字符读入并减去 '0' 后，累加到 sum 中，每次累加前执行 sum *= 10，代码中 sum 暂时存放在寄存器 $t5 中，最后把 $t5 的值 move 至 $v0 中并返回

(2) 自定义 write_int 函数部分代码：

```
1                   la      $t0,    temp_char_arr       #address of arr
2                   li      $t1,    0                   #length
3                   li      $t2,    0                   #index
4                   li      $t3,    10                  #divisor
5
6                   loop:
7
8                   div     $a0,    $t3                 #quotient in lo, remainder in hi
9                   mflo    $a0                         #a0 /= 10
10                  mfhi    $t5                         #t5 = a0 % 10
11                  addi    $t5,    '0'                 #num to char
12                  add     $t4,    $t0,    $t1         #t4 = t0+i
13                  addi    $t1,    1                   #++i
14                  sb      $t5,    ($t4)               #save remainder in a0+i
15                  beqz    $a0,    init_reverse        #finish decomposing the integer
16                  b       loop
17
```

思路：将整数拆解成一个个字符存入额外的数组，调整顺序后进行输出
需要输出的整数作为变量由 $a0 传入函数，loop 中每次将 $a0 mod 10 的值存入额外数组，然后使 $a0 /= 10，循环直至 $a0 == 0；

```
1                   init_reverse:
2
3                   srl     $t5,    $t1,    1           #t5 = length / 2
4                   addi    $t1,    $t1,    -1          #--t1 for convenience
5
6                   reverse_arr:
7
8                   bge     $t2,    $t5,    print_arr   #if idx >= length / 2, goto print_arr
9                   sub     $t3,    $t1,    $t2         #t3 = length - 1 - idx
10                  move    $t4,    $t2                 #save idx
11
12                  add     $t2,    $t0,    $t2         #t2 = addr + idx
13                  add     $t3,    $t0,    $t3         #t3 = addr + length - 1 - idx
14
15                  lb      $t6,    ($t2)               #temp1 = arr[i]
16                  lb      $t7,    ($t3)               #temp2 = arr[j]
17                  sb      $t7,    ($t2)               #arr[i] = temp2
```

```
                sb      $t6,    ($t3)               #arr[j] = temp1

                move    $t2,    $t4
                addi    $t2,    $t2,    1           #++idx
                b       reverse_arr

```

用上述方法拆解的整数存入数组时是逆序的（即原本整数为 123，则数组存储 “321”），因此需要将字符串 reverse，每次将 temp[i] 与 temp[length-1-i] 的值进行交换，直至 i >= length/2

最后将所得数组输出，则可实现 write_int 功能

(3) 实现 sort 功能

思路，用选择排序对读入的整数数组进行排序

```
        sort:
        li      $t3,    9                       #selection sort: i < len - 1

        loop_i:

        init_i:
        li      $t1,    0                       #i = 0

        check_i:
        bge     $t1,    $t3,    init_write_int_arr  #i >= len - 1, end loop_i

        loop_j:

        init_j:
        move    $t2,    $t1
        addi    $t2,    $t2,    1               #j = i + 1

        check_j:
        bge     $t2,    $t6,    increase_i      #j >= len, end loop_j
        addi    $sp,    $sp,    -8              #sorry, I am too stupid to implement it with 9
regs,
                                                #so I have to ask for more space TAT

        major_part:
        sll     $t5,    $t1,    2
        add     $t5,    $t7,    $t5
        sw      $t5,    ($sp)                   #store arr + i
        lw      $t5,    ($t5)                   #t5 = arr[i]

        sll     $t8,    $t2,    2
        add     $t8,    $t7,    $t8
        sw      $t8,    4($sp)                  #store arr + j
        lw      $t8,    ($t8)                   #t8 = arr[j]

        ble     $t5,    $t8,    increase_j      #arr[i] <= arr[j], do not swap
        #swap
        move    $t9,    $t5
        lw      $t5,    ($sp)
        sw      $t8,    ($t5)
        lw      $t8,    4($sp)
        sw      $t9,    ($t8)
```

```
41
42          increase_j:
43          addi    $t2,    $t2,    1
44          addi    $sp,    $sp,    8
45          b       check_j                         #j++, and then check j if >= len
46
47          increase_i:
48          addi    $t1,    $t1,    1
49          b       check_i                         #i++, and the check i if >= len - 1
50
51
```

解析：上述代码用 mips 汇编实现二重循环，结构如下：

```
init_i:                                 //i = 0
check_i:                                //i < length - 1
    init_j:                             //j = i + 1
    check_j:                            //j < length
          major_part: swap();           //do swapping here
    increase_j:                         //j++
increase_i:                             //i++
```

(4) makefile

在实验过程中发现，mips 交叉编译过程需要输入多条命令，因此编写 makefile 文件

```
assembler=mips-linux-gnu-as
linker=mips-linux-gnu-ld
src=sort.S
obj=ex.o
dest=ex



$(dest):$(obj)
    $(linker) -g $(obj) -e main -o ./$(dest)

$(obj):
    $(assembler) -g $(src) -o $(obj)



.PHONY : clean
clean :
    rm $(obj) $(dest)
```

(5) 实验结果：

```
chs@ubuntu:~/MIPSexp$ make
mips-linux-gnu-as -g sort.S -o ex.o
mips-linux-gnu-ld -g ex.o -e main -o ./ex
chs@ubuntu:~/MIPSexp$ qemu-mips ./ex
4 1 3 1 6 5 17 9 8 6
1 1 3 4 5 6 6 8 9 17
chs@ubuntu:~/MIPSexp$ qemu-mips ./ex
2341 64 675
  5
0
9999 7698
    114514
  1919         77777
0 5 64 675 1919 2341 7698 9999 77777 114514
```

实验结果 1_1

a. 解析：

i. 对于样例数据的正常输入能得到正确的结果；

ii. 当输入带有空格符以及换行符时，能正确执行整数的读入

```
chs@ubuntu:~/MIPSexp$ qemu-mips ./ex
2147483647 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 2147483647
```

实验结果 2_1

```
chs@ubuntu:~/MIPSexp$ qemu-mips ./ex
2147483648
qemu: unhandled CPU exception 0x15 - aborting
pc=0x0040028c HI=0x00000000 LO=0x7ffffff8 ds 00a2 00000000 0
GPR00: r0 00000000 at 00000000 v0 00000001 v1 00000000
GPR04: a0 00000000 a1 004104e4 a2 00000001 a3 00000000
GPR08: t0 00000038 t1 00000039 t2 0000000a t3 00000030
GPR12: t4 00000008 t5 7ffffff8 t6 0000000a t7 00410505
GPR16: s0 00000000 s1 00000000 s2 00000000 s3 00000000
GPR20: s4 00000000 s5 00000000 s6 00000000 s7 00000000
GPR24: t8 00410505 t9 00000000 k0 00000000 k1 00000000
```

实验结果 2_2

b. 解析：

mips 的寄存器为 32 位，当输入数据大于 2 的 31 次方-1，即 2147483647 时，数据将产生错误，因此需要控制输入的大小

五、 实验心得

(1) 在实验开始时，阅读实验环境指导后，仍然对实验环境要求一知半解，没有深究便去编写代码，以为 syscall 可以直接使用处理器调用（然而只允许使用 Linux 内核所提供的系统调用），因此导致实验初问题不断，后来在询问助教以及查阅有关资料后终于稍微明白了实验环境的说明，因此以后阅读实验环境指导时，有不懂的地方需要先去查找相关资料了解，以免造成不必要的时间上的浪费；

(2) 在本次实验中，因为不熟悉 misp 汇编的格式，编写代码时，.data 段的声明误加了逗号，导致编译器没有为变量分配空间，所有变量共用一个地址，debug 了相当长一段时间都没有发现，后来在助教的帮助下终于发现了问题，因此以后进行计组实验时，一定要细心；

(3) 通过本次实验，我初步认识和掌握了 MIPS 汇编语言程序设计的基本方法，熟悉了 QEMU 模拟器和 GNU/Linux 下 x86-MIPS 交叉编译环境的使用，对 linux 的内核调用有了一定的了解。

【程序代码】存放在 MIPSexp 文件夹中，使用时先进入 MIPSexp 文件夹，输入

```
~/MIPSexp$    make
```

生成可执行代码，再输入

```
~/MIPSexp$    qemu-mips ./ex
```

调用程序。