俞楚凡 ICS 3.24 Lab 实验报告

任务一中，我需要分别比较在C语言中嵌入汇编读取条件码判断是否进位，溢出等等的方法与使用普通的判断方法（正数加和为负之类）的效率优劣。

```
check_carry:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    %edx, 24(%rbp)
    movl    16(%rbp), %edx
    movl    24(%rbp), %eax
    addl    %edx, %eax
    cmpl    %eax, 16(%rbp)
    seta    %al
    movzbl  %al, %eax
    popq    %rbp
    ret
    .seh_endproc
    .globl  check_overflow
    .def    check_overflow; .scl    2;  .type   32; .endef
    .seh_proc   check_overflow
```

上面为使用正常方法判断的汇编代码。可以看到，这种方法的效率较低，在一个函数中需要进行大量的操作。

```
get_carry_flag:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $16, %rsp
    .seh_stackalloc 16
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    %edx, 24(%rbp)
    movl    16(%rbp), %eax
    movl    24(%rbp), %edx
```

而直接读取机器码的方式相对效率很高。

但是也有一个缺点，即，直接读取机器码的方式可移植性较差，因为其涉及到了系统底层的运行机理。而使用C语言直接判断的方法虽然效率较差，但是其通用性较强，换到其他系统上也能正常运行。

第二个任务中，要求在不使用编译优化开关的情况下，比较for累加写法和while累加写法的异同。

使用以下命令生成`circulate.s`文件：

```
gcc -S circulate.c
```

查看`circulate.s`文件：

```
sum_for:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $16, %rsp
    .seh_stackalloc 16
    .seh_endprologue
    movq    %rcx, 16(%rbp)
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    jmp .L2
```

上述为sum_for函数的汇编代码图示。

```
sum_while:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $16, %rsp
    .seh_stackalloc 16
    .seh_endprologue
    movq    %rcx, 16(%rbp)
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    jmp .L8
```

上述为sum_while函数的汇编代码图示。

可以看到，二者几乎没有任何区别。事实上，在汇编层级，循环本质上都是使用条件测试和跳转语句组合的，而for循环和while循环本质上会先变成类似do while循环的形式，最终得到相似的结构。

使用下列命令加入-O1优化开关：

```
gcc -S -O1 circulate.c -o circulateO1.s
```

打开`circulateO1.s`：

```
sum_for:
    .seh_endprologue
    leaq    4096(%rcx), %r8
    addq    $4198400, %rcx
    movl    $0, %eax
    jmp .L2
```

首先看到，编译器直接删去了栈帧设置。其次，往下可以看到下面的跳转表的长度也有了一定程度上的减少，同样是减少了堆栈操作，而直接使用寄存器进行计算，提高了效率。

```
sum_for:
    .seh_endprologue
    xorl    %eax, %eax
    leaq    4096(%rcx), %r8
    addq    $4198400, %rcx
.L2:
    leaq    -4096(%r8), %rdx
    .p2align 4,,10
.L3:
    addl    (%rdx), %eax
    addq    $4, %rdx
    cmpq    %r8, %rdx
    jne .L3
    leaq    4096(%rdx), %r8
    cmpq    %rcx, %r8
    jne .L2
    ret
    .seh_endproc
    .p2align 4,,15
    .globl  sum_while
    .def    sum_while;  .scl    2;  .type   32; .endef
    .seh_proc   sum_while
```

而O2相比O1来讲，代码更加简短，编译器直接将初始化和跳转合并，并且引入了p2align这样的对齐指令，并且使用了更紧凑的控制流，提高了效率。

```
sum_for:
    .seh_endprologue
    xorl    %eax, %eax
    leaq    4096(%rcx), %r8
    addq    $4198400, %rcx
.L2:
    leaq    -4096(%r8), %rdx
    pxor    %xmm0, %xmm0
    .p2align 4,,10
.L3:
    movdqu  (%rdx), %xmm2
    addq    $16, %rdx
    cmpq    %rdx, %r8
    paddd   %xmm2, %xmm0
    jne .L3
    movdqa  %xmm0, %xmm1
    addq    $4096, %r8
    psrldq  $8, %xmm1
    paddd   %xmm1, %xmm0
    movdqa  %xmm0, %xmm1
    psrldq  $4, %xmm1
    paddd   %xmm1, %xmm0
    movd    %xmm0, %edx
    addl    %edx, %eax
    cmpq    %rcx, %r8
    jne .L2
    ret
```

而O3相比O2做出了更多激进的优化。引入了SIMD指令进行向量化计算（xmm寄存器)，可以并行计算，减少循环次数；使用movdqu指令加载未对齐的数据，提升了内存访问效率；减少了循环次数从而减少了分支预测的开销，大幅提高了效率。

在第三个任务中，我需要根据给定的代码，生成汇编文件，并观察跳转表。

`switch.s`中存储了生成的汇编代码。

```
.L4:
    .long   .L10-.L4
    .long   .L2-.L4
    .long   .L9-.L4
    .long   .L8-.L4
    .long   .L7-.L4
    .long   .L6-.L4
    .long   .L5-.L4
    .long   .L3-.L4
    .long   .L3-.L4
    .text
```

这一段是跳转表的核心。.L4使用.long指令定义了跳转表，右侧的每一行分别代表了一个地址相对偏移量。当程序访问到了跳转表时，程序会根据这个偏移量寻找到指定的分支。

```
    .file   "switch.c"
    .text
    .def    __main; .scl    2;  .type   32; .endef
    .section .rdata,"dr"
    .align 8
```

这一段定义了程序的元数据。

```
.LC0:
    .ascii "please input x(ranges from 100 to 110):\0"
.LC1:
    .ascii "%d\0"
.LC2:
    .ascii "the res is %d\0"
    .text
    .globl  main
    .def    main;   .scl    2;  .type   32; .endef
    .seh_proc   main
main:
```

这一段定义了一些字符串常量，并且定义了main函数。汇编程序采用数据段与代码段分离的方式，也即，虽然printf和scanf语句被写在main函数内，其中的字符串存储在main函数之外。

```
main:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $48, %rsp
    .seh_stackalloc 48
    .seh_endprologue
    call    __main
    movl    $10, -8(%rbp)
    movl    $0, -4(%rbp)
    leaq    .LC0(%rip), %rcx
    call    puts
    leaq    -8(%rbp), %rax
    movq    %rax, %rdx
    leaq    .LC1(%rip), %rcx
    call    scanf
    movl    -8(%rbp), %eax
    subl    $100, %eax
    cmpl    $8, %eax
    ja  .L2
    movl    %eax, %eax
    leaq    0(,%rax,4), %rdx
    leaq    .L4(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L4(%rip), %rdx
    addq    %rdx, %rax
    jmp *%rax
    .section .rdata,"dr"
    .align 4
```

这一段，首先是执行了输入输出，其次通过`jmp`语句对于%rax寄存器的值的判断进行跳转，实现switch分支语句。

```
.L10:
    movl    $1, -4(%rbp)
    jmp .L11
.L9:
    movl    $2, -4(%rbp)
    jmp .L11
.L8:
    movl    $3, -4(%rbp)
.L7:
    addl    $4, -4(%rbp)
.L6:
    addl    $5, -4(%rbp)
    jmp .L11
.L5:
    movl    $6, -4(%rbp)
    jmp .L11
.L3:
    movl    $7, -4(%rbp)
    jmp .L11
.L2:
    movl    $10, -4(%rbp)
.L11:
    movl    -4(%rbp), %eax
    movl    %eax, %edx
    leaq    .LC2(%rip), %rcx
    call    printf
    movl    $0, %eax
    addq    $48, %rsp
```

从跳转表的实现我们可以看到，事实上加上了break的分支是直接跳转到最后一个.L11，也即之后的语句，而不加break的分支则是顺序往下执行的。

使用下列命令生成目标文件：

```
gcc -c switch.c -o switch.o
```

使用`objdump`进行反汇编：

```
objdump -d switch.o
```

得到反汇编结果如下。

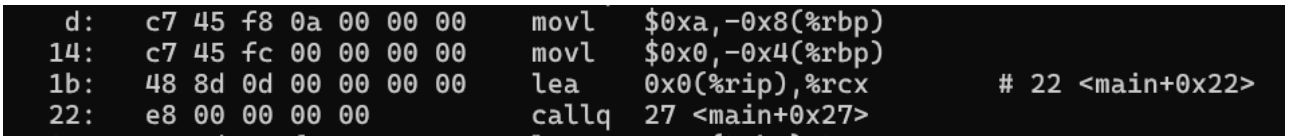
Disassembly of section .text:

0000000000000000

: 0: 55 push %rbp 1: 48 89 e5 mov %rsp,%rbp 4: 48 83 ec 30 sub $0x30,%rsp 8: e8 00 00 00 00 callq d <main+0xd> d: c7 45 f8 0a 00 00 00 movl $0xa,-0x8(%rbp) 14: c7 45 fc 00 00 00 00 movl $0x0,-0x4(%rbp) 1b: 48 8d 0d 00 00 00 00 lea 0x0(%rip),%rcx # 22 <main+0x22> 22: e8 00 00 00 00 callq 27 <main+0x27> 27: 48 8d 45 f8 lea -0x8(%rbp),%rax 2b: 48 89 c2 mov %rax,%rdx 2e: 48 8d 0d 28 00 00 00 lea 0x28(%rip),%rcx # 5d <main+0x5d> 35: e8 00 00 00 00 callq 3a <main+0x3a> 3a: 8b 45 f8 mov -0x8(%rbp),%eax 3d: 83 e8 64 sub $0x64,%eax 40: 83 f8 08 cmp $0x8,%eax 43: 77 57 ja 9c <main+0x9c> 45: 89 c0 mov %eax,%eax 47: 48 8d 14 85 00 00 00 lea 0x0(,%rax,4),%rdx 4e: 00 4f: 48 8d 05 3c 00 00 00 lea 0x3c(%rip),%rax # 92 <main+0x92> 56: 8b 04 02 mov (%rdx,%rax,1),%eax 59: 48 98 cltq 5b: 48 8d 15 3c 00 00 00 lea 0x3c(%rip),%rdx # 9e <main+0x9e> 62: 48 01 d0 add %rdx,%rax 65: ff e0 jmpq *%rax 67: c7 45 fc 01 00 00 00 movl $0x1,-0x4(%rbp) 6e: eb 33 jmp a3 <main+0xa3> 70: c7 45 fc 02 00 00 00 movl $0x2,-0x4(%rbp) 77: eb 2a jmp a3 <main+0xa3> 79: c7 45 fc 03 00 00 00 movl $0x3,-0x4(%rbp) 80: 83 45 fc 04 addl $0x4,-0x4(%rbp) 84: 83 45 fc 05 addl $0x5,-0x4(%rbp) 88: eb 19 jmp a3 <main+0xa3> 8a: c7 45 fc 06 00 00 00 movl $0x6,-0x4(%rbp) 91: eb 10 jmp a3 <main+0xa3> 93: c7 45 fc 07 00 00 00 movl $0x7,-0x4(%rbp) 9a: eb 07 jmp a3 <main+0xa3> 9c: c7 45 fc 0a 00 00 00 movl $0xa,-0x4(%rbp) a3: 8b 45 fc mov -0x4(%rbp),%eax a6: 89 c2 mov %eax,%edx a8: 48 8d 0d 2b 00 00 00 lea 0x2b(%rip),%rcx # da <main+0xda> af: e8 00 00 00 00 callq b4 <main+0xb4> b4: b8 00 00 00 00 mov $0x0,%eax b9: 48 83 c4 30 add $0x30,%rsp bd: 5d pop %rbp be: c3 retq bf: 90 nop

初始化栈帧。

调用了一个地址未解析的函数。



初始化了一个变量，加载了一个字符串常量，并调用了函数。

```
27:   48 8d 45 f8            lea    -0x8(%rbp),%rax
2b:   48 89 c2               mov    %rax,%rdx
2e:   48 8d 0d 28 00 00 00   lea    0x28(%rip),%rcx        # 5d <main+0x5d>
35:   e8 00 00 00 00         callq  3a <main+0x3a>
```

调用了三次函数，由此可以推得分别对应程序中的printf，scanf和switch。

```
3a:   8b 45 f8               mov    -0x8(%rbp),%eax
3d:   83 e8 64               sub    $0x64,%eax
40:   83 f8 08               cmp    $0x8,%eax
43:   77 57                  ja     9c <main+0x9c>
```

如果输入不在100-108内，则跳转到末尾。否则，依次处理可能的合法情况。

```
a3:   8b 45 fc               mov    -0x4(%rbp),%eax
a6:   89 c2                  mov    %eax,%edx
a8:   48 8d 0d 2b 00 00 00   lea    0x2b(%rip),%rcx        # da <main+0xda>
af:   e8 00 00 00 00         callq  b4 <main+0xb4>
b4:   b8 00 00 00 00         mov    $0x0,%eax
b9:   48 83 c4 30            add    $0x30,%rsp
bd:   5d                     pop    %rbp
be:   c3                     retq
bf:   90                     nop
```

最后打印结果，释放栈帧，结束程序。

```
65:   ff e0                  jmpq   *%rax
67:   c7 45 fc 01 00 00 00   movl   $0x1,-0x4(%rbp)
6e:   eb 33                  jmp    a3 <main+0xa3>
70:   c7 45 fc 02 00 00 00   movl   $0x2,-0x4(%rbp)
77:   eb 2a                  jmp    a3 <main+0xa3>
79:   c7 45 fc 03 00 00 00   movl   $0x3,-0x4(%rbp)
80:   83 45 fc 04            addl   $0x4,-0x4(%rbp)
84:   83 45 fc 05            addl   $0x5,-0x4(%rbp)
88:   eb 19                  jmp    a3 <main+0xa3>
8a:   c7 45 fc 06 00 00 00   movl   $0x6,-0x4(%rbp)
91:   eb 10                  jmp    a3 <main+0xa3>
93:   c7 45 fc 07 00 00 00   movl   $0x7,-0x4(%rbp)
9a:   eb 07                  jmp    a3 <main+0xa3>
9c:   c7 45 fc 0a 00 00 00   movl   $0xa,-0x4(%rbp)
```

程序中的switch语句对应这段跳转表。