计算机体系架构　Lab02

范云潜 18373486

微电子学院 184111 班

日期：2020 年 10 月 9 日

Problem ex1

对于一个 32-bit 的整数 `x`，加法发生溢出的边界是 `~x`，此时恰好得到 `~0x0`，若是加数大于此边界（无符号类型）即可判断发生了溢出。

Listing 1: ex1 solution

```
# if the op1 and op2 hava different sign
# there will be no carry out.
# {op1_sign, op2_sign, ans_sign} = {1, 1, 0} or {0, 0, 1}
# in 2 instructions
# the largest op2 for op1 is ~op1 (unsigned)
# so we need to compare op2 and ~op1
# the 'not x' could be 'x xor 0xffffffff'

# test 1 passed

# li $t3, 0x80000000
# li $t4, 0x7fffffff

# test 2 passed

# li $t3, 0x00000000
# li $t4, 0xffffffff

# test 3 passed

# li $t3, 0x00000001
# li $t4, 0xffffffff

xori $t3, $t3, 0xffffffff
sltu $t2, $t3, $t4
```

Problem ex2

对于一个浮点数，当其越来越大，每两个数字之间的间距也会越来越大，那么我们首先挑选一个很大的浮点数[1] {0b 0 | 111 1111 0 | 000 0000 0000 0000 0000 = 170141183460469231731687303715884105728.000

[1]通过 C 进行转换成浮点数

= 0x7f000000} 。通过 C 和 MARS 进行实验，发现的确如此。

Problem ex3

此处需要找到一个**最小的**浮点数，因此需要仔细考虑浮点数的间隔下界大于 1 的临界点，此时作为尾数域的最小变化会引起整个浮点数增加 2 ，那么指数是 24 = 0x18 ，那么指数域为 24+127 = 0x97 ，那么位表示为 {0b 0 | 100 1011 1 | 000 0000 0000 0000 0000 = 16777216.000000 = 0x4b800000} 。若是尝试 {0x4b800000 - 1= 0x4b7fffff = 16777215.000000} 则会失败，验证完毕。

Problem ex4

事实上浮点数的不可交换原因之一是浮点数的特殊值也就是 ∞ 和 NaN 。如果前两个值相加会溢出，而第三个值的提前出现会防止这种情况的出现。这样的集合可以表示为

$$\{a, b, c | a + b \text{ overflow } \&\& |a + c| < |a| \text{ thus } (a + c) + b \text{ don't overflow}\}$$

进行举例，最大的规格化值 {0b 0 1111 1110 1111 1111 1111 1111 1111 111 } ，它的一半 {0b 0 1111 1101 1111 1111 1111 1111 1111 111} ，它的相反数 {0b 1 1111 1111 1110 1111 1111 1111 1111 111 } 分别作为 a, b, c 。

Listing 2: ex4 solution-1

```c
#include <stdio.h>
#include <inttypes.h>

int main()
{

    unsigned a = 0x7f7fffffu;
    float* ap = (float*)(&a);
    unsigned b = 0x7effffffu;
    float* bp = (float*)(&b);
    unsigned c = 0xfeffffffu;
    float* cp = (float*)(&c);
    printf("%f\n%f\n%f\n%f\n%f\n", (*ap), *bp, *cp, (*ap + *bp + *cp), (*ap + *
        cp + *bp));
}
```

输出为

```
    340282346638528859811704183484516925440.000000
    170141173319264429905852091742258462720.000000
    -170141173319264429905852091742258462720.000000
    inf
    340282346638528859811704183484516925440.000000
```

另一方面的原因还包括，对于同一符号类型的数据，由于阶数不同，相邻浮点数的间隔也不同，若是求和时顺序使得被加数在间隔内，则不会引起变化。定义浮点数 a 的阶数下，尾数变化的最小单位为 δ_a 。那么 $(a + b) + c \neq a + (b + c)$ 可以表述为

$$\{a, b, c | a \geq \delta_b \&\& a < \delta_{b+c}\}$$

举例, b 和 c 取间隔为 2 的浮点数 {0b 0 | 100 1011 1 | 000 0000 0000 0000 0000 = 16777216.000000 = 0x4b800000} ， a 取浮点数 2.1 。

Listing 3: ex4 solution-2

```
#include <stdio.h>

int main()
{
    float b = 16777216.000000;
    float c = b;
    float a = 2.1;
    int* ap = (int*)(&a);

    printf("%x\n%f\n%f\n", *ap, (a+b)+c, a+(b+c));
    return 0;
}
```

输出为

```
    40066666
    33554432.000000
    33554436.000000
```

进一步的，需要进行 MIPS 的程序设计，请见 p2_1.s 。

Listing 4: ex4 solution-2 mips

```
  .data
  # This shows you can use a .word and directly encode the value in hex
  # if you so choose
num1: .word 0x4b800000 # 16777216.000000
num2: .float 2.1
num0:   .float 0.0
result: .word 0
string: .asciiz "\n"
string1: .asciiz "(a+b)+c is "
string2: .asciiz "a+(b+c) is "

  .text
main:
  la $t0, num1
  lwc1 $f2, 0($t0) # num1 b
  lwc1 $f3, 0($t0) # num1 c
  lwc1 $f4, 4($t0) # num2 a

  # Print out the values of the summands
```

```
  # num1
li $v0, 2
mov.s $f12, $f2
syscall

  # \n
li $v0, 4
la $a0, string
syscall

  # num2
li $v0, 2
mov.s $f12, $f4
syscall

  # \n
li $v0, 4
la $a0, string
syscall

  # (a+b)+c
li $v0, 4
la $a0, string1
syscall
  # a+b
add.s $f12, $f2, $f4
  # +c
add.s $f12, $f12, $f3

# Transfer the value from the floating point reg to the integer reg

swc1 $f12, 8($t0)
lw $s0, 8($t0)

# At this point, $f12 holds the sum, and $s0 holds the sum which can
# be read in hexadecimal

  # print ans in f12
li $v0, 2
syscall



li $v0, 4
la $a0, string
syscall
```

```
    # a+(b+c)
  li $v0, 4
  la $a0, string2
  syscall
    # c+b
  add.s $f12, $f2, $f3
    # +a
  add.s $f12, $f12, $f4

  # Transfer the value from the floating point reg to the integer reg

  swc1 $f12, 8($t0)
  lw $s0, 8($t0)

  # At this point, $f12 holds the sum, and $s0 holds the sum which can
  # be read in hexadecimal

    # print ans in f12
  li $v0, 2
  syscall


  li $v0, 4
  la $a0, string
  syscall





  # This jr crashes MARS
  # jr $ra
```

输出为

```
    1.6777216E7
    2.1
    (a+b)+c is 3.3554432E7
    a+(b+c) is 3.3554436E7

-- program is finished running (dropped off bottom) --
```