PA1 实验报告

刘时宜 201180078

2022 年 3 月 15 日

目录

1 PA1-1

参考实验说明，在nemu/include/cpu/reg.h中修改CPU_STATE的定义即可，其中将一些原有代码中的struct关键字改成了union关键字。修改如下（与实验说明中相同）：

```
typedef struct
{
    union {
        union {
            union {
                uint32_t _32;
                uint16_t _16;
                uint8_t _8[2];
            };
            uint32_t val;
        } gpr[8];
        struct { // do not change the order of the registers
            uint32_t eax, ecx, edx, ebx, esp, ebp, esi, edi;
        };
    };
} CPU_STATE;
```

修改后在~/pa_nju目录下执行make clean、make编译项目，然后make test_pa-1执行测试用例，可以看到reg_test() pass的提示，说明CPU寄存器模拟成功

1.1 思考题

C语言中的struct和union关键字都是什么含义，寄存器结构体的参考实现为什么把部分struct改成了union？

struct关键字在中文中的翻译为“结构体”，union关键字在中文中的翻译为“联合体”。两者都是c语言中复杂数据结构类型的实现方式。不同点在于，struct中定义的各基本数据占用的是不同的内存空间，不同数据互不干扰。而union中定义的各基本数据占用的是相同的内存空间，数据最低位对齐，整个union数据结构所占内存空间与其中占内存最大的数据所用的内存相同。

由CPU中寄存器的定义方式，例如eax寄存器的低16位称为ax，ax的低8位称为al，可知eax、ax、al、ah本质上占用的是一块内存空间，相互之间有影响，与union中各数据类型相互影响的定义相同，故应该采用union关键字进行定义。

2 PA1-2

2.1 add

参照给出的样例实现代码，填入alu_add中。

```
uint32_t alu_add(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t res = 0;
    res = dest + src;

    set_CF_add(res, dest, data_size);
    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);
    set_OF_add(res, src, dest, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));

}
```

然后，同样写出其中引用的函数。

```
inline void set_ZF(uint32_t result, size_t data_size){
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.ZF = (result == 0);
}

inline void set_SF(uint32_t result, size_t data_size){
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.SF = sign(result);
}

inline void set_PF(uint32_t result){
    int count = 0;
    result = result & (0xFF);

    while(result){
        count += result & 0x1;
        result = result >> 1;
    }

    cpu.eflags.PF = !(count % 2);
}

inline void set_CF_add(uint32_t result, uint32_t src, size_t data_size){
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    cpu.eflags.CF = result < src;
}

inline void set_OF_add(uint32_t result, uint32_t src, uint32_t dest, size_t
    data_size){
    switch(data_size){
        case 8:
            result = sign_ext(result & 0xFF, 8);
            src = sign_ext(src & 0xFF, 8);
            dest = sign_ext(dest & 0xFF, 8);
```

```
35              break;
36          case 16:
37              result = sign_ext(result & 0xFFFF, 16);
38              src = sign_ext(src & 0xFFFF, 16);
39              dest = sign_ext(dest & 0xFFFF, 16);
40              break;
41          default: break;
42      }
43
44
45      if(sign(src) == sign(dest)){
46          if(sign(src) != sign(result))
47              cpu.eflags.OF = 1;
48          else
49              cpu.eflags.OF = 0;
50      }
51      else{
52          cpu.eflags.OF = 0;
53      }
54  }
```

`make test_pa-1`后通过了测试样例。

2.2 adc

带进位的加法与add中的实现几乎相同，除了判断CF标志位的时候有所改变。原先由于变量大小关系，若有进位结果一定小于原先的操作数。有进位输入之后，产生进位输出时结果也可能等于原先的操作数。因此要根据原先CF位的值做不同判断。

```
1  inline void set_CF_adc(uint32_t result, uint32_t src, uint32_t cf, size_t
       data_size){
2      result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
3      src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
4      if(cf)
5          cpu.eflags.CF = result <= src;
6      else
7          cpu.eflags.CF = result < src;
8  }
9
10 uint32_t alu_adc(uint32_t src, uint32_t dest, size_t data_size){
11     // printf("src: %08x, dest: %08x, CF: %01x, data_size: %d\n", src, dest, cpu.
       eflags.CF, data_size);
12     uint32_t res = 0;
13     res = dest + src + cpu.eflags.CF;
14     // printf("res: %08x\n", res);
15
16     set_CF_adc(res, dest, cpu.eflags.CF, data_size);
17     set_PF(res);
18     set_ZF(res, data_size);
```

```
19      set_SF(res, data_size);
20      set_OF_add(res, src, dest, data_size);
21
22      return res & (0xFFFFFFFF >> (32 - data_size));
23  }
```

2.3 sub

无借位减法sbb的实现，除了更改运算符号为“-”以外，还要特别处理CF和OF标志位的处理函数。

CF标志位设置函数中吧CF值的表达式改为`cpu.eflags.CF = result > dest;`，吧小于号改为大于号即可。

OF标志位在将减数取补码的基础上，绝大多数代码沿用add中OF标志位设置函数的代码即可。唯一要注意的一点是机器数全为1的数补码为其自身，此时取补码后简单沿用相加的判定标准会出错。将这部分做特殊情况处理即可，即`invalid complement`相关代码部分。

```
1   inline void set_CF_sub(uint32_t result, uint32_t dest, size_t data_size){
2       // printf("In set_CF_sub:\n");
3       result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
4       dest = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);
5       cpu.eflags.CF = result > dest;
6       // printf("    dest: %08x, result: %08x, data_size: %d\n", dest, result,
        data_size);
7       // printf("    CF: %d\n", cpu.eflags.CF);
8   }
9
10  void set_OF_sub(uint32_t result, uint32_t src, uint32_t dest, size_t data_size){
11      //printf("In set_OF_sub:\n");
12      int invalid_complement = 0;
13      switch(data_size){
14          case 8:
15              result = sign_ext(result & 0xFF, 8);
16              src = sign_ext(src & 0xFF, 8);
17              dest = sign_ext(dest & 0xFF, 8);
18              if(src == 0x80)
19                  invalid_complement = 1;
20              break;
21          case 16:
22              result = sign_ext(result & 0xFFFF, 16);
23              src = sign_ext(src & 0xFFFF, 16);
24              dest = sign_ext(dest & 0xFFFF, 16);
25              if(src == 0x8000)
26                  invalid_complement = 1;
27              break;
28          default:
29              if(src == 0x80000000)
```

```
30                  invalid_complement = 1;
31              break;
32      }
33      //printf("    dest: %08x, result: %08x, data_size: %d\n", dest, result,
        data_size);
34
35      if(invalid_complement){
36          if(!sign(dest)){//dest >= 0, certainly overflow
37              cpu.eflags.OF = 1;
38              //printf("    OF: 1, invalid complement.\n");
39              return;
40          }
41          else{
42              cpu.eflags.OF = 0;
43              //printf("    OF: 0, invalid complement.\n");
44              return;
45          }
46      }
47
48      src = ~src + 1;
49      if(sign(src) == sign(dest)){
50          if(sign(dest) != sign(result)){
51              cpu.eflags.OF = 1;
52              //printf("    OF: 1\n");
53          }
54          else{
55              cpu.eflags.OF = 0;
56              //printf("    OF: 0\n");
57          }
58      }
59      else{
60          cpu.eflags.OF = 0;
61          //printf("    OF: 0\n");
62      }
63  }
64
65  uint32_t alu_sub(uint32_t src, uint32_t dest, size_t data_size){
66      //printf("dest: %08x, src: %08x, data_size: %d\n", dest, src, data_size);
67      uint32_t res = 0;
68      res = dest - src;
69      //printf("res: %08x\n", res);
70
71      set_CF_sub(res, dest, data_size);
72      set_PF(res);
73      set_ZF(res, data_size);
74      set_SF(res, data_size);
75      set_OF_sub(res, src, dest, data_size);
76
77      return res & (0xFFFFFFFF >> (32 - data_size));
78  }
```

2.4 sbb

sbb函数中，在sub函数的基础上，额外减去借位即可。需要额外处理的标志位仍然是CF和OF。

`set_CF_sbb`函数中，仍然需要根据借位的情况，使用大于或者大于等于的判断标准，与`set_CF_adc`函数中类似。

`set_OF_sbb`函数中，除了减数本身的补码表示可能是自身外，还要考虑借位后补码不正常的影响，需要多一些特殊情况的分类与处理。

```
inline void set_CF_sbb(uint32_t result, uint32_t dest, uint32_t cf, size_t
    data_size){
    // printf("In set_CF_sub:\n");
    result = sign_ext(result & (0xFFFFFFFF >> (32 - data_size)), data_size);
    dest = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);
    if(cf)
        cpu.eflags.CF = (result >= dest);
    else
        cpu.eflags.CF = (result > dest);
    // printf("    dest: %08x, result: %08x, data_size: %d\n", dest, result,
    data_size);
    // printf("    CF: %d\n", cpu.eflags.CF);
}

void set_OF_sbb(uint32_t result, uint32_t src, uint32_t dest, uint32_t cf, size_t
     data_size){
    //printf("In set_OF_sub:\n");
    int invalid_complement = 0;
    switch(data_size){
        case 8:
            result = sign_ext(result & 0xFF, 8);
            src = sign_ext(src & 0xFF, 8);
            dest = sign_ext(dest & 0xFF, 8);
            if(src == 0x80)
                invalid_complement = 1;
            break;
        case 16:
            result = sign_ext(result & 0xFFFF, 16);
            src = sign_ext(src & 0xFFFF, 16);
            dest = sign_ext(dest & 0xFFFF, 16);
            if(src == 0x8000)
                invalid_complement = 1;
            break;
        default:
            if(src == 0x80000000)
                invalid_complement = 1;
            break;
    }
    //printf("    dest: %08x, result: %08x, data_size: %d\n", dest, result,
    data_size);
```

```
37
38      if(invalid_complement){
39          if(!cf){
40              if(!sign(dest)){//dest >= 0, certainly overflow
41                  cpu.eflags.OF = 1;
42                  //printf("    OF: 1, invalid complement, cf = 0\n");
43                  return;
44              }
45              else{
46                  cpu.eflags.OF = 0;
47                  //printf("    OF: 0, invalid complement, cf = 0\n");
48                  return;
49              }
50          }
51          else{
52              if(!sign(dest) && (dest != 0)){//dest >= 0, certainly overflow
53                  cpu.eflags.OF = 1;
54                  //printf("    OF: 1, invalid complement, cf = 1\n");
55                  return;
56              }
57              else{
58                  cpu.eflags.OF = 0;
59                  //printf("    OF: 0, invalid complement, cf = 1\n");
60                  return;
61              }
62          }
63      }
64
65      src = ~src + 1 - cf;
66      //printf("    dest: %08x, result: %08x, data_size: %d\n", dest, result,
        data_size);
67      if(sign(src) == sign(dest)){
68          if(sign(dest) != sign(result)){
69              cpu.eflags.OF = 1;
70              //printf("    OF: 1\n");
71          }
72          else{
73              cpu.eflags.OF = 0;
74              //printf("    OF: 0\n");
75          }
76      }
77      else{
78          cpu.eflags.OF = 0;
79          //printf("    OF: 0\n");
80      }
81  }
82
83  uint32_t alu_sbb(uint32_t src, uint32_t dest, size_t data_size){
84      //printf("\ndest: %08x, src: %08x, CF: %d, data_size: %d\n", dest, src, cpu.
        eflags.CF, data_size);
```

```
85      uint32_t res = 0;
86      res = dest - src - cpu.eflags.CF;
87      //printf("res: %08x\n", res);
88
89      set_OF_sbb(res, src, dest, cpu.eflags.CF, data_size);
90      set_CF_sbb(res, dest, cpu.eflags.CF, data_size);
91      set_PF(res);
92      set_ZF(res, data_size);
93      set_SF(res, data_size);
94
95      return res & (0xFFFFFFFF >> (32 - data_size));
96  }
```

2.5 mul

将两操作数根据符号位将高位置零，求出真值后相乘即可得到运算结果，需要注意的是相乘的时候需要显式类型转换，否则将在乘法运算后截断得到`uint32_t`类型的结果后，才转换成`uint64_t`，产生结果错误。

根据手册说明，CF和OF依据比`data_size`位数高的各位是否为0就可以设置了。

```
1  uint64_t alu_mul(uint32_t src, uint32_t dest, size_t data_size){
2      // printf("dest: %x, src: %x, data_size: %d\n", dest, src, data_size);
3      uint64_t res = 0;
4      src = unsign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
5      dest = unsign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);
6
7      res = (uint64_t)src * (uint64_t)dest;
8      // printf("res: %llx\n", res);
9
10     cpu.eflags.OF = cpu.eflags.CF = !((res >> data_size) == 0);
11     // printf("(res >> data_size): %llx\n", (res >> data_size));
12     // printf("OF: %d, CF: %d\n", cpu.eflags.OF, cpu.eflags.CF);
13
14     return res;
15 }
```

2.6 imul

与`alu_mul`函数基本相同，乘法时只需要先将数据转换为带符号的int类型后利用C语言本身的带符号整数相乘就可以得到结果了。

OF、CF位的设置只需要看res的低datasize位的数学真值是否与res本身的数学真值相同即可判定。

```
1  int64_t alu_imul(int32_t src, int32_t dest, size_t data_size){
2      // printf("dest: %x, src: %x, data_size: %d\n", dest, src, data_size);
3      int64_t res = 0;
```

```
    src = sign_ext(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    dest = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    res = (int64_t)src * (int64_t)dest;
    // printf("res: %llx\n", res);

    cpu.eflags.OF = cpu.eflags.CF = (res == sign_ext(res & (0xFFFFFFFF >> (32 - data_size)), data_size));
    // printf("(res >> data_size): %llx\n", (res >> data_size));
    // printf("OF: %d, CF: %d\n", cpu.eflags.OF, cpu.eflags.CF);

    return res;
}
```

2.7 div mod

根据datasize对操作数进行无符号扩展，然后相除、求余即可。根据约定，除数为0时直接抛出错误并结束程序即可。

```
uint32_t alu_div(uint64_t src, uint64_t dest, size_t data_size){
    //printf("dest: %llx, src: %llx, data_size: %d\n", dest, src, data_size);
    if(src == 0){
        printf("Error: Divided by 0\n");
        assert(src != 0);
    }

    uint32_t res = 0;
    src = unsign_ext_64(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    dest = unsign_ext_64(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    res = dest / src;
    //printf("res: %x\n", res);
    //getchar();

    return res;
}

uint32_t alu_mod(uint64_t src, uint64_t dest){
    if(src == 0){
        printf("Error: Divided by 0\n");
        assert(src != 0);
    }

    uint32_t res = 0;

    res = dest % src;

    return res;
}
```

为了实现无符号扩展，编写函数unsign_ext_64并放在pa_nju/nemu/include/cpu/alu.h中。

```
inline uint64_t unsign_ext_64(uint32_t x, size_t data_size){
    assert(data_size == 16 || data_size == 8 || data_size == 32);
        switch (data_size)
        {
        case 8:
                return (uint64_t)(x & 0xff);
        case 16:
                return (uint64_t)(x & 0xffff);
        default:
                return (uint64_t)x;
        }
}
```

2.8 idiv imod

只需要将无符号除法、求模的代码中无符号扩展的部分改为有符号扩展即可。

```
int32_t alu_idiv(int64_t src, int64_t dest, size_t data_size){
    //printf("dest: %llx, src: %llx, data_size: %d\n", dest, src, data_size);
    if(src == 0){
        printf("Error: Divided by 0\n");
        assert(src != 0);
    }

    int32_t res = 0;
    src = sign_ext_64(src & (0xFFFFFFFF >> (32 - data_size)), data_size);
    dest = sign_ext_64(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    res = dest / src;
    //printf("res: %x\n", res);
    //getchar();

    return res;
}

int32_t alu_imod(int64_t src, int64_t dest){
    if(src == 0){
        printf("Error: Divided by 0\n");
        assert(src != 0);
    }

    int32_t res = 0;

    res = dest % src;

    return res;
}
```

2.9 and or xor

做按位运算后设置标志位即可。

```
uint32_t alu_and(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t res = 0;
    res = src & dest;

    cpu.eflags.OF = 0;
    cpu.eflags.CF = 0;
    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}

uint32_t alu_xor(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t res = 0;
    res = src ^ dest;

    cpu.eflags.OF = 0;
    cpu.eflags.CF = 0;
    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}

uint32_t alu_or(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t res = 0;
    res = src | dest;

    cpu.eflags.OF = 0;
    cpu.eflags.CF = 0;
    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}
```

2.10 shl sal

参照i386手册上的伪代码，将其用c语言描述即可

```
uint32_t alu_shl(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t temp = src;
    uint32_t res = 0;
```

```
    res = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    while(temp!=0){
        cpu.eflags.CF = sign(res);
        res = res << 1;
        res = sign_ext(res & (0xFFFFFFFF >> (32 - data_size)), data_size);
        temp--;
    }

    if(src == 1){
        cpu.eflags.OF = (cpu.eflags.CF != sign(res));
    }

    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}

uint32_t alu_sal(uint32_t src, uint32_t dest, size_t data_size){
    uint32_t temp = src;
    uint32_t res = 0;
    res = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    while(temp!=0){
        cpu.eflags.CF = sign(res);
        res = res << 1;
        res = sign_ext(res & (0xFFFFFFFF >> (32 - data_size)), data_size);
        temp--;
    }

    if(src == 1){
        cpu.eflags.OF = (cpu.eflags.CF != sign(res));
    }

    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}
```

2.11 shr

由于是逻辑右移，在使用C语言右移功能前将类型转换为无符号类型后右移，按说明操作标志位即可。

需要注意的一点是res的符号才操作过程中可能发生了改变，需要提前判断OF的设置情

况。

```
uint32_t alu_shr(uint32_t src, uint32_t dest, size_t data_size){
    // printf("dest: %x, count: %d, data_size: %d\n", dest, src, data_size);
    uint32_t temp = src;
    uint32_t res = 0;
    res = unsign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    if(src == 1){
        cpu.eflags.OF = sign(res);
    }

    while(temp!=0){
        cpu.eflags.CF = res & 0x1;
        res = ((unsigned int)res) >> 1;
        res = sign_ext(res & (0xFFFFFFFF >> (32 - data_size)), data_size);
        temp--;
    }

    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    return res & (0xFFFFFFFF >> (32 - data_size));
}
```

2.12 sar

与shr基本相同，只不过在右移前转换为有符号整型即可。

由于做的是算数右移，符号不会改变，故可以在右移结束后再设置OF位。

```
uint32_t alu_sar(uint32_t src, uint32_t dest, size_t data_size){
    // printf("dest: %x, count: %d, data_size: %d\n", dest, src, data_size);
    uint32_t temp = src;
    uint32_t res = 0;
    res = sign_ext(dest & (0xFFFFFFFF >> (32 - data_size)), data_size);

    while(temp!=0){
        cpu.eflags.CF = res & 0x1;
        // printf("res: %x\n", res);
        res = ((int32_t)res) >> 1;
        // printf("shifted res: %x\n", res);
        res = sign_ext(res & (0xFFFFFFFF >> (32 - data_size)), data_size);
        temp--;
    }

    if(src == 1){
        cpu.eflags.OF = (cpu.eflags.CF != sign(res));
    }

```

```
    set_PF(res);
    set_ZF(res, data_size);
    set_SF(res, data_size);

    // getchar();
    return res & (0xFFFFFFFF >> (32 - data_size));
}
```

3 PA1-3

3.1 加减法

只需要完成移位量的计算即可。唯一需要注意的是两数可能是非规格化数或规格化数，需要分别讨论。

```
/* TODO: shift = ? */
if(fb.exponent == 0){ // fa, fb both unnormalized
    shift = fb.exponent - fa.exponent;
    //printf("    fa, fb both unnormalized, shift: %d\n", shift);
}
else if(fa.exponent == 0){  //fb is normalized, fa is unnormalized
    shift = fb.exponent - fa.exponent - 1;
    //printf("    fb is normalized, fa is unnormalized, shift: %d\n", shift);
}
else{  // fa, fb both normalized
    shift = fb.exponent - fa.exponent;
    //printf("    fa, fb both normalized, shift: %d\n", shift);
}
```

3.2 乘除法

唯一需要完成的是阶码的计算，参照实验指导说明填写即可。

```
/* TODO: exp_res = ? leave space for GRS bits. */
exp_res = fa.exponent + fb.exponent - 127 - 20;
```

3.3 规范化 internal_normalize

首先看保护位的舍入。按照规则进行舍入即可。产生进位后还要处理可能的溢出或非规格数转规格数等等情况。偷懒不写专门的处理代码了，直接使用`goto`语句跳转到函数开始处再完整进行一次处理即可。

```
inline uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs){
    GRS_OVERFLOW:;

    ...
```

```
5
6   if (!overflow)
7   {
8     /* TODO: round up and remove the GRS bits */
9     //printf("    round up and remove the GRS bits\n");
10    uint32_t grs = sig_grs & 0x7;
11    //printf("        grs: %d\n", grs);
12    if(grs < 4){
13        sig_grs = sig_grs >> 3;
14        //printf("        remove the GRS bits\n");
15    }
16    else if(grs == 4){
17        sig_grs = sig_grs >> 3;
18        if(sig_grs & 0x1){ //sig_grs & 0x1 == 1
19            sig_grs++;
20            sig_grs = sig_grs << 3;
21            //printf("        round up GRS bits, goto GRS_OVERFLOW\n");
22            goto GRS_OVERFLOW;
23        }
24        //printf("        remove the GRS bits\n");
25    }
26    else{
27        sig_grs = sig_grs >> 3;
28        sig_grs++;
29        sig_grs = sig_grs << 3;
30        //printf("        round up GRS bits, goto GRS_OVERFLOW\n");
31        goto GRS_OVERFLOW;
32    }
33  }
34    ...
35 }
```

其余按照注释给出的指令完成即可。需要注意的是规格化数与非规格化数的转换过程中不需要增减阶码，与一般的左移或右移不同。

完整的代码如下：

```
1  inline uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs){
2      //printf("In internal_normalize:\n");
3      GRS_OVERFLOW:;
4      //printf("    sign: %d, exp: %d, sig_grs: %016llx\n", sign, exp, sig_grs);
5
6      // normalization
7      bool overflow = false; // true if the result is INFINITY or 0 during
       normalize
8
9      if ((sig_grs >> (23 + 3)) > 1 || exp < 0)
10     {
11         // normalize toward right
12         //printf("    normalize toward right\n");
13         while ((((sig_grs >> (23 + 3)) > 1) && exp < 0xff) // condition 1
```

```
14              ||                          // or
15              (sig_grs > 0x04 && exp < 0)           // condition 2
16              )
17      {
18          /* TODO: shift right, pay attention to sticky bit*/
19          bool sticky = 0;
20          sticky = sticky | (sig_grs & 0x1);
21          sig_grs = sig_grs >> 1;
22          sig_grs |= sticky;
23          exp++;
24          //printf("        Shift right: 0\n");
25          //printf("        sign: %d, exp: %d, sig_grs: %016llx\n", sign, exp,
    sig_grs);
26      }
27
28      if (exp >= 0xff)
29      {
30          /* TODO: assign the number to infinity */
31          exp = 0xff;
32          sig_grs = 0;
33          overflow = true;
34      }
35      if (exp == 0)
36      {
37          // we have a denormal here, the exponent is 0, but means 2^-126,
38          // as a result, the significand should shift right once more
39          /* TODO: shift right, pay attention to sticky bit*/
40          bool sticky = 0;
41          sticky = sticky | (sig_grs & 0x1);
42          sig_grs = sig_grs >> 1;
43          sig_grs |= sticky;
44      }
45      if (exp < 0)
46      {
47          /* TODO: assign the number to zero */
48          exp = 0;
49          sig_grs = 0;
50          overflow = true;
51      }
52  }
53  else if (((sig_grs >> (23 + 3)) == 0) && exp > 0)
54  {
55      // normalize toward left
56      //printf("    normalize toward left\n");
57      while (((sig_grs >> (23 + 3)) == 0) && exp > 0)
58      {
59          sig_grs = sig_grs << 1;
60          exp--;
61          //printf("        shift left: 1\n");
62          //printf("        sign: %d, exp: %d, sig_grs: %016llx\n", sign, exp,
```

```
    sig_grs);
63      }
64      if (exp == 0)
65      {
66          // denormal
67          /* TODO: shift right, pay attention to sticky bit*/
68          bool sticky = 0;
69          sticky = sticky | (sig_grs & 0x1);
70          sig_grs = sig_grs >> 1;
71          sig_grs |= sticky;
72          //printf("        Denormal shift right: 1\n");
73          //printf("        sign: %d, exp: %d, sig_grs: %016llx\n", sign, exp,
    sig_grs);
74      }
75  }
76  else if (exp == 0 && sig_grs >> (23 + 3) == 1)
77  {
78      // two denormals result in a normal
79      exp++;
80      //printf("    two denormals result in a normal\n");
81  }
82
83  if (!overflow)
84  {
85      /* TODO: round up and remove the GRS bits */
86      //printf("    round up and remove the GRS bits\n");
87      uint32_t grs = sig_grs & 0x7;
88      //printf("        grs: %d\n", grs);
89      if(grs < 4){
90          sig_grs = sig_grs >> 3;
91          //printf("        remove the GRS bits\n");
92      }
93      else if(grs == 4){
94          sig_grs = sig_grs >> 3;
95          if(sig_grs & 0x1){ //sig_grs & 0x1 == 1
96              sig_grs++;
97              sig_grs = sig_grs << 3;
98              //printf("        round up GRS bits, goto GRS_OVERFLOW\n");
99              goto GRS_OVERFLOW;
100         }
101         //printf("        remove the GRS bits\n");
102     }
103     else{
104         sig_grs = sig_grs >> 3;
105         sig_grs++;
106         sig_grs = sig_grs << 3;
107         //printf("        round up GRS bits, goto GRS_OVERFLOW\n");
108         goto GRS_OVERFLOW;
109     }
110 }
```

```
111
112     FLOAT f;
113     f.sign = sign;
114     f.exponent = (uint32_t)(exp & 0xff);
115     f.fraction = sig_grs; // here only the lowest 23 bits are kept
116     //printf("     Result: %08x\n", f.val);
117     return f.val;
118 }
```

3.4 思考题

1）对应输入是规格化或非规格化数，而输出产生了阶码上溢结果为正（负）无穷的情况

当两数均为浮点数能表示的最大数时，相加或相乘肯定产生溢出。可以写一段代码验证：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main(){
5      union{
6          unsigned int val;
7          float fval;
8      }a,b,sum, mul;
9
10     a.val = 0x7f7fffff;
11     b.val = 0x7f7fffff;
12
13     sum.fval = a.fval + b.fval;
14     mul.fval = a.fval * b.fval;
15
16     printf("a.fval: %f\n", a.fval);
17     printf("b.fval: %f\n\n", b.fval);
18
19     printf("sum.fval: %f\n", sum.fval);
20     printf("mul.fval: %f\n", mul.fval);
21 }
```

运行结果：

```
a.fval: 340282346638528860000000000000000000000.000000
b.fval: 340282346638528860000000000000000000000.000000

sum.fval: 1.#INF00
mul.fval: 1.#INF00
```

2）对应输入是规格化或非规格化数，而输出产生了阶码下溢结果为正（负）零的情况。

加法时，只需要令两数为相反数即可。

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

void main(){
    union{
        unsigned int val;
        float fval;
    }a,b,sum, mul;

    a.val = 0x7f7fffff;
    b.val = 0xff7fffff;

    sum.fval = a.fval + b.fval;

    printf("a.fval: %f\n", a.fval);
    printf("b.fval: %f\n\n", b.fval);

    printf("sum.fval: %f\n", sum.fval);
    printf("sum.val: %08x\n", sum.val);
}
```

运行结果：

```
a.fval: 340282346638528860000000000000000000000.000000
b.fval: -340282346638528860000000000000000000000.000000

sum.fval: 0.000000
sum.val: 00000000
```

乘法时，令两数的阶码均极小，使得阶码下溢即可。

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

void main(){
    union{
        unsigned int val;
        float fval;
    }a,b,sum, mul;

    a.val = 0x80001;
    b.val = 0x80001;

    mul.fval = a.fval * b.fval;

    printf("a.fval: %e\n", a.fval);
    printf("b.fval: %e\n\n", b.fval);

    printf("mul.fval: %e\n", mul.fval);
    printf("mul.val: %08x\n", mul.val);
```

```
20 }
```

运行结果：

```
a.fval: 7.346854e-040
b.fval: 7.346854e-040

mul.fval: 0.000000e+000
mul.val: 00000000
```

4 心得体会

1. 指导手册与上课讲的内容对理解框架代码的实现原理很有帮助！很大程度上帮我克服了阅读理解源代码的恐惧感。

2. 感觉参考资料稍微有点多，而且每个资料都只有含有部分信息？例如在guide里的对函数功能的注释不在源代码里，有些提示只在ppt里出现等等，找的时候稍微有点乱。

3. 在写代码之前一定要先明确代码的功能是什么，有哪些情况，应该怎么实现，即先确保代码的正确性，然后再开始写。否则将在debug的过程中花费的精力远多于一开始提高代码的正确性所用的理论推导时间。例如在`alu_sbb`函数的编写中就陷入了不停的debug写bug的过程，不如一开始就想好有哪些情况，怎样实现最简洁。