

计算机系统基础
Programming Assignment

PA 1-3 – 浮点数的表示和运算

2020年9月24日 / 2020年9月25日
南京大学《计算机系统基础》课程组

中秋国庆放假通知

- 10 月 10 日（星期六）上 10 月 8 日（星期四）的课

一	二	三	四	五	六	日
28 十二	29 十三	30 十四	休 1 国庆节	休 2 十六	休 3 十七	休 4 十八
休 5 十九	休 6 二十	休 7 廿一	休 8 寒露	9 廿三	班 10 廿四	11 廿五

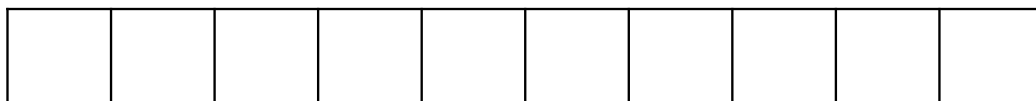
前情提要

PA 1-1

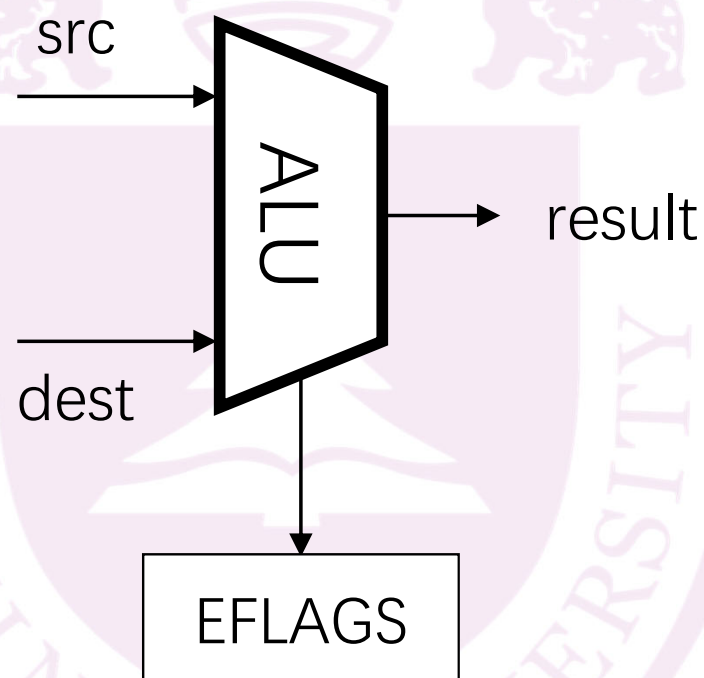
通用寄存器

EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI

主存 (RAM)



PA 1-2



目录

- PA 1-1 数据的类型和存取
- PA 1-2 整数的表示和运算
- PA 1-3 浮点数的表示和运算



概览

PA 1-1

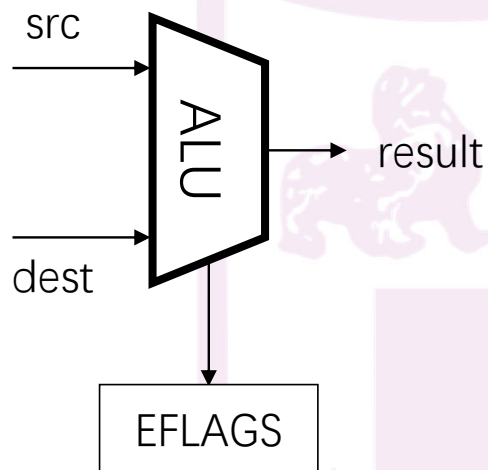
通用寄存器

EAX
ECX
EDX
EBX
ESP
EBP
ESI
EDI

主存 (RAM)



PA 1-2



PA 1-3



x87

<http://www.felixcloutier.com/x86/>

带小数的实数

3.75

(十进制)

11.11

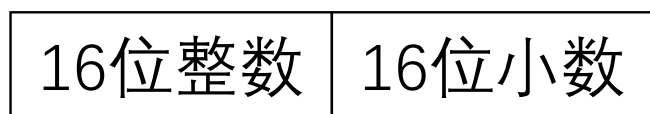
(二进制)

2^1 2^0 2^{-1} 2^{-2}

定点数和浮点数

带小数的实数
(真值)

定点数 (机器数)



自定义一种32位
定点数表示法

小数点

浮点数 (机器数)



IEEE 754

定点数和浮点数

11.11

带小数的实数
(真值)

定点数 (机器数)



自定义一种32位
定点数表示法

小数点

00...00111100...00

14个0

14个0

浮点数 (机器数)



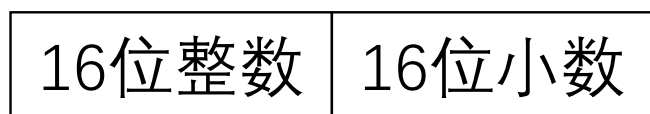
IEEE 754

定点数和浮点数

11.11

带小数的实数
(真值)

定点数 (机器数)



自定义一种32位
定点数表示法

小数点

00...00111100...00

14个0

14个0

浮点数 (机器数)



IEEE 754

010000000011100...0

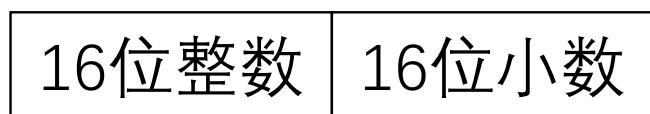
$$11.11 = 1.111 \times 2^1$$

定点数和浮点数

11.11

带小数的实数
(真值)

定点数 (机器数)



自定义一种32位
定点数表示法

小数点

00...00111100...00

14个0

14个0

浮点数 (机器数)



IEEE 754

01000000011100...0

11.11 = 1.111 x 2¹

IEEE 754 浮点数标准

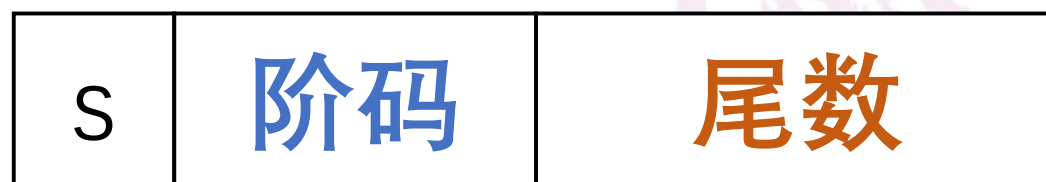


William Kahan (1933 - ?)
ACM Turing Award, 1989

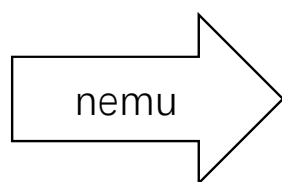
For his fundamental contributions to numerical analysis. One of the foremost experts on floating-point computations. Kahan has dedicated himself to "making the world safe for numerical computations"!

本课程只谈表示和运算，不谈理论

IEEE 754 浮点数标准



IEEE 754



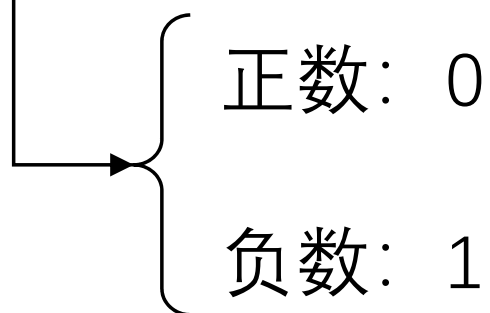
32位单精度浮点数: float
64位双精度浮点数: double

IEEE 754 浮点数标准

1位



IEEE 754



(类比带符号整数原码表示法)

IEEE 754 浮点数标准



IEEE 754

阶码
数值

偏置
常数

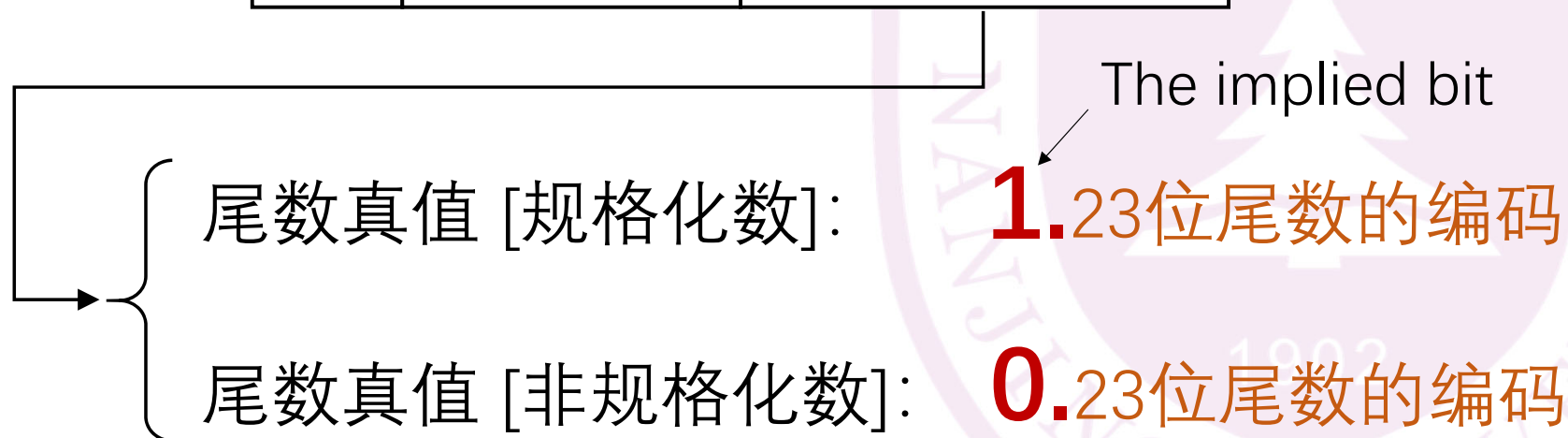
非零 [规格化数]: 表示 $2^{\text{exp} - 127}$

全零 [非规格化数]: 表示 2^{-126}

IEEE 754 浮点数标准



IEEE 754



IEEE 754 浮点数标准



IEEE 754

尾数真值 [规格化数]:

1.23位尾数的编码

尾数真值 [非规格化数]:

0.23位尾数的编码

nemu变量起名: 带上隐藏位的 significand

不带隐藏位的 fraction

IEEE 754 浮点数标准

表 2.2 IEEE754 浮点数的解释

值的类型	单精度(32 位)				双精度(64 位)			
	符号	阶码	尾数	值	符号	阶码	尾数	值
正零	0	0	0	0	0	0	0	0
负零	1	0	0	-0	1	0	0	-0
正无穷大	0	255(全 1)	0	∞	0	2047(全 1)	0	∞
负无穷大	1	255(全 1)	0	$-\infty$	1	2047(全 1)	0	$-\infty$
无定义数 (非数)	0 或 1	255(全 1)	$\neq 0$	NaN	0 或 1	2047(全 1)	$\neq 0$	NaN
规格化非零正数	0	$0 < e < 255$	f	$2^{e-127}(1.f)$	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
规格化非零负数	1	$0 < e < 255$	f	$-2^{e-127}(1.f)$	1	$0 < e < 2047$	f	$-2^{e-1023}(1.f)$
非规格化正数	0	0	$f \neq 0$	$2^{-126}(0.f)$	0	0	$f \neq 0$	$2^{-1022}(0.f)$
非规格化负数	1	0	$f \neq 0$	$-2^{-126}(0.f)$	1	0	$f \neq 0$	$-2^{-1022}(0.f)$

IEEE 754 浮点数标准

nemu/include/cpu/reg_fpu.h

```
typedef union {  
    struct // IEEE 754 float结构  
    {  
        uint32_t fraction : 23;  
        uint32_t exponent : 8;  
        uint32_t sign : 1;  
    };  
    float fval;    // 浮点数  
    uint32_t val;  // 机器数  
} FLOAT;
```



浮点数的运算模拟

- NEMU中模拟浮点数的算术运算的部件

- FPU - 浮点运算单元（大部分由框架代码实现）

浮点栈
状态字
控制字

- 实现浮点数运算：加减乘除

- 相关代码：[nemu/src/cpu/fpu.c](#)

- 需要实现`internal_float_xxx()`函数

- xxx可以是add、sub、mul或div

- 需要实现`internal_normalize()`函数

浮点数的运算模拟

[nemu/src/cpu/fpu.c](#)

```
uint32_t internal_float_add(uint32_t b, uint32_t a);
```

浮点数的运算模拟

运算名称

```
uint32_t internal_float_add(uint32_t b, uint32_t a);
```

运算结果

操作数b

操作数a

浮点数的运算模拟

运算名称

```
uint32_t internal_float_add(uint32_t b, uint32_t a);
```

运算结果

操作数b

操作数a

float对应的机器数, IEEE 754


浮点数的运算模拟

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{

}
}
```

浮点数的运算模拟（禁用方法）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    FLOAT fa, fb, f;
    fa.val = a;
    fb.val = b;
    f.fval = fa.fval + fb.fval;
    return f.val;
}
```



但可以用来测试

浮点数的运算模拟（要求的方法）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    利用位操作和整数运算来实现浮点数运算
    Soft Floating Point
}
```

浮点数的运算模拟（要求的方法）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    利用位操作和整数运算来实现浮点数运算
    Soft Floating Point
}
```

参考: <https://bellard.org/softfp/>

传奇程序员: QEMU、FFMPEG

浮点数的运算模拟（基本流程）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```

浮点数的运算模拟（以加法为例）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
```

1. 处理边界情况（NaN、0、INF）
2. 提取符号、阶码、尾数
3. 整数运算得到中间结果
4. 舍入并规格化后返回

```
}
```

框架代码已经针对浮点数的加减乘除运算完成了对边界情况的处理

```
CORNER_CASE_RULE corner_add[] = {
    {P_ZERO_F, P_ZERO_F, P_ZERO_F},
    {N_ZERO_F, P_ZERO_F, P_ZERO_F},
    ...
} // nemu/src/cpu/fpu.c
```

浮点数的运算模拟

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```

提取符号、阶码、尾数

nemu/src/cpu/fpu.c

```
uint32_t internal_float_add(uint32_t b, uint32_t a) {
    FLOAT f, fa, fb;
    fa.val = a;
    fb.val = b;
    ...
    uint32_t sig_a, sig_b, sig_res;
    sig_a = fa.fraction;
    if (fa.exponent != 0)
        sig_a |= 0x800000;
    sig_b = fb.fraction;
    if (fb.exponent != 0)
        sig_b |= 0x800000;
    ...
}
```

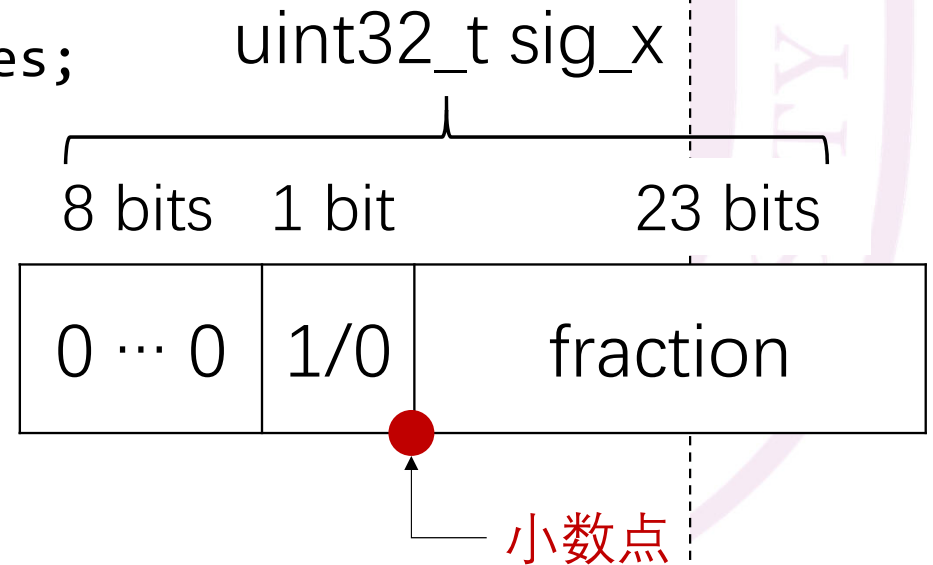
nemu/include/cpu/reg_fpu.h

```
typedef union {
    struct // IEEE 754 float结构
    {
        uint32_t fraction : 23;
        uint32_t exponent : 8;
        uint32_t sign : 1;
    };
    float fval; // 浮点数
    uint32_t val; // 机器数
} FLOAT;
```

提取符号、阶码、尾数

[nemu/src/cpu/fpu.c](#)

```
uint32_t internal_float_add(uint32_t b, uint32_t a) {  
    FLOAT f, fa, fb;  
    fa.val = a;  
    fb.val = b;  
  
    ...  
    uint32_t sig_a, sig_b, sig_res;    uint32_t sig_x  
    sig_a = fa.fraction;  
    if (fa.exponent != 0)  
        sig_a |= 0x800000;  
    sig_b = fb.fraction;  
    if (fb.exponent != 0)  
        sig_b |= 0x800000;  
  
    ...  
}
```



浮点数的运算模拟

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```


整数运算得到中间结果

- 浮点数做加法（减法）的步骤

1. 对阶：小阶向大阶看齐

2. 尾数相加（相减）

$$1.1 \times 2^{10} + 1.11 \times 2^{12}$$

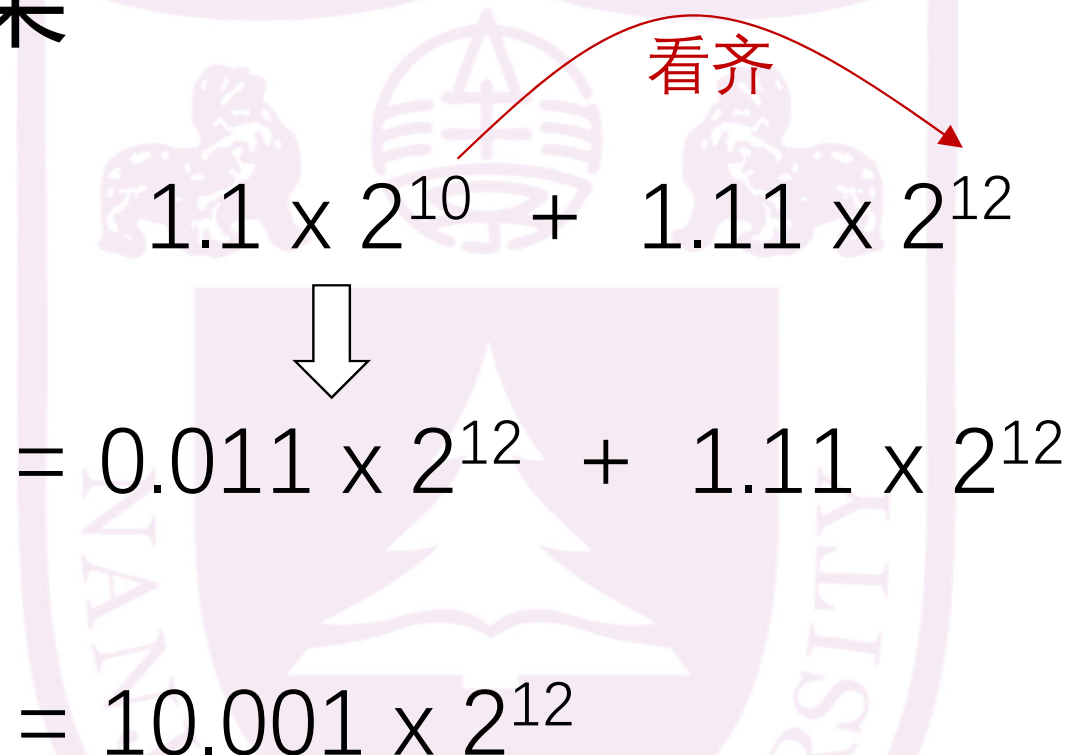
整数运算得到中间结果

- 浮点数做加法（减法）的步骤

1. 对阶：小阶向大阶看齐

小阶增加至大阶，同时尾数右移，保证对应真值不变

2. 尾数相加（相减）


$$\begin{aligned} & 1.1 \times 2^{10} + 1.11 \times 2^{12} \\ & \quad \downarrow \\ & = 0.011 \times 2^{12} + 1.11 \times 2^{12} \\ & = 10.001 \times 2^{12} \end{aligned}$$

```
uint32_t internal_float_add(uint32_t b, uint32_t a) {
```

```
    FLOAT f, fa, fb;
```

```
    fa.val = a;
```

```
    fb.val = b;
```

```
    ...
```

```
    if (fa.exponent > fb.exponent) {
```

```
        fa.val = b;
```

```
        fb.val = a;
```

```
    }
```

```
    // alignment shift for fa
```

```
    uint32_t shift = 0;
```

```
    /* TODO: shift = ? */
```

```
    printf("\e[0;31mPlease implement me at fpu.c\e[0m\n");
```

```
    assert(0);
```

```
    assert(shift >= 0);
```

```
    ...
```

```
}
```

对阶：小阶向大阶看齐

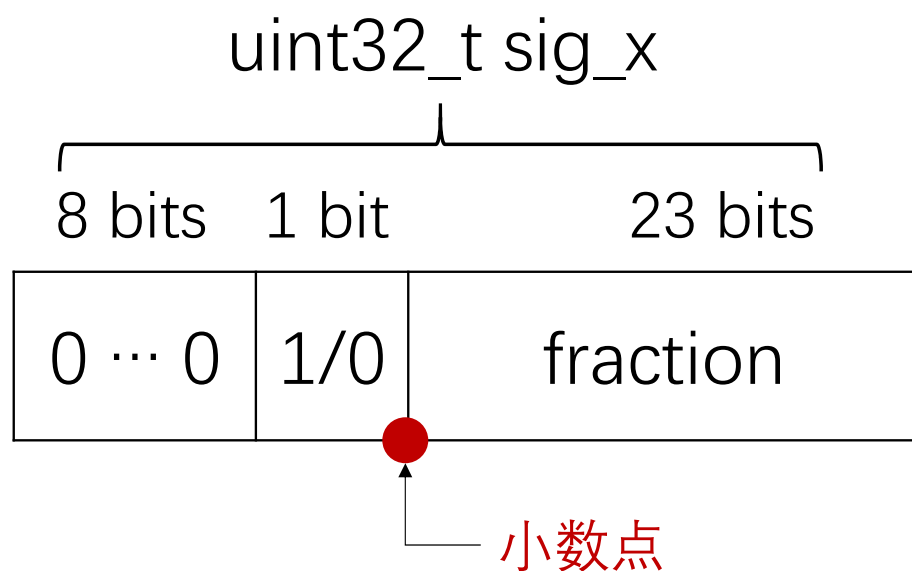
fa中保留阶较小的数

fb中保留阶较大的数

将移位的位数计算出来，将阶较小的数的尾数部分右移，计算shift时注意非规格化数的情形

[nemu/src/cpu/fpu.c](#)

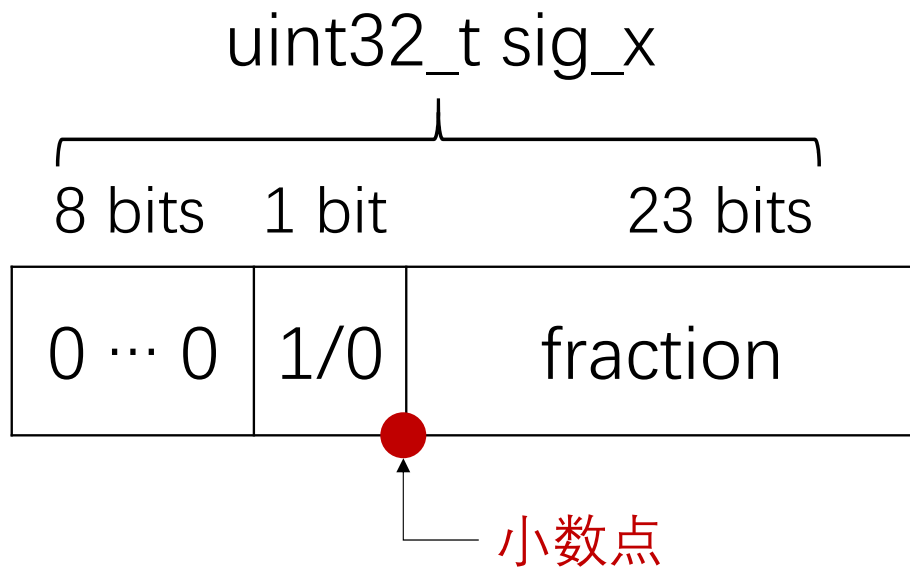
对阶：小阶向大阶看齐



尾数（含隐藏位）

>> shift

对阶：小阶向大阶看齐

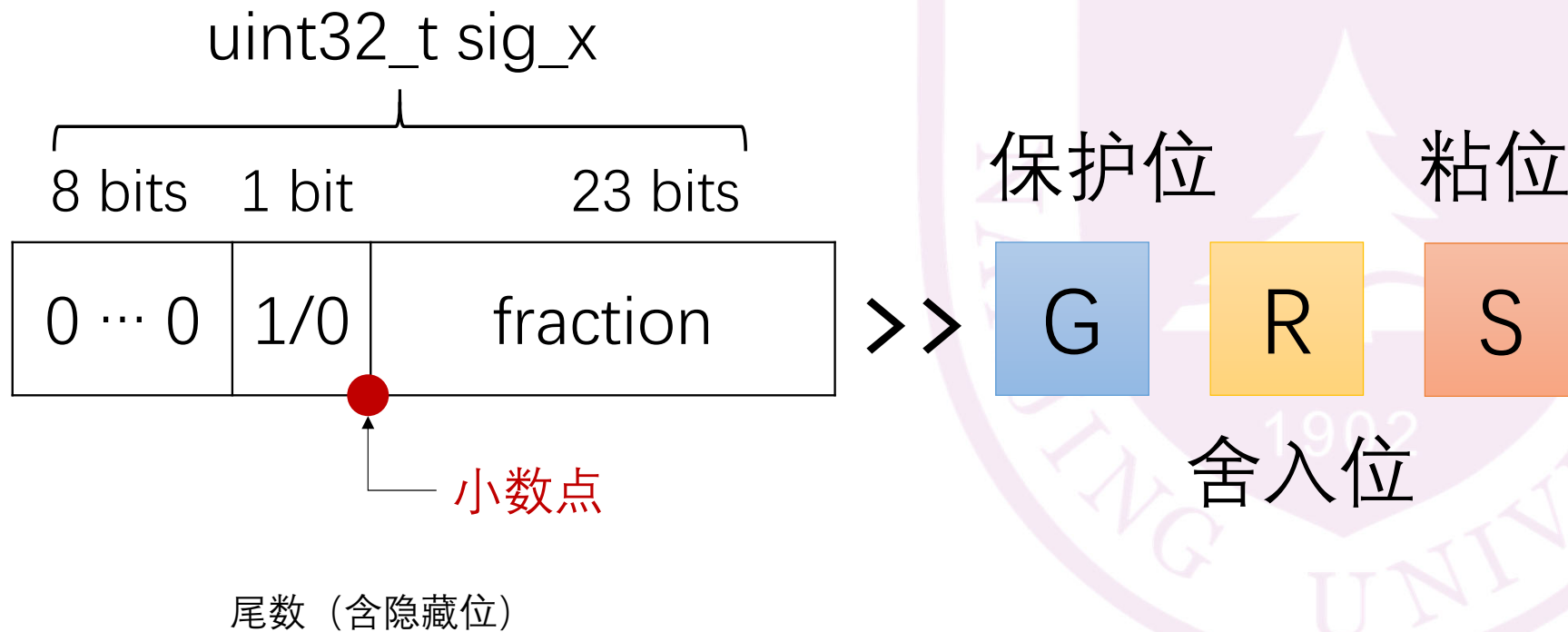


尾数（含隐藏位）

丢弃移出的位？精度的严重损失
shift ≥ 24 ?

>> shift

对阶：小阶向大阶看齐



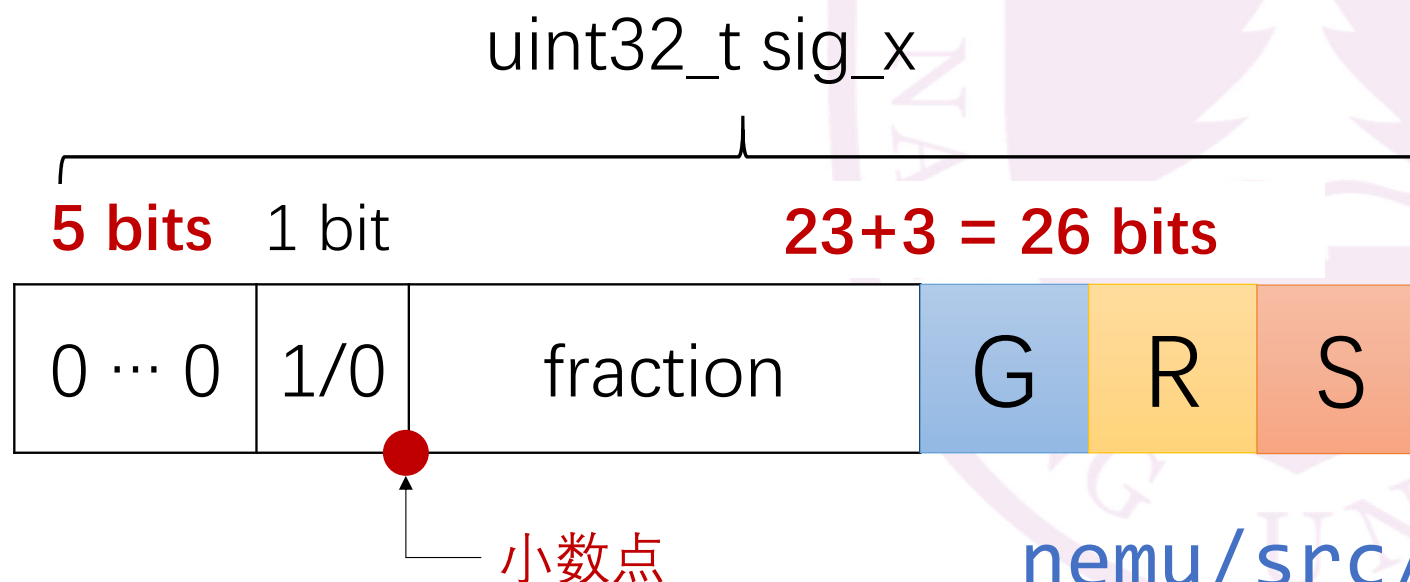
```

uint32_t internal_float_add(uint32_t b, uint32_t a) {
    FLOAT f, fa, fb;
    fa.val = a;
    fb.val = b;
    ...
    sig_a = (sig_a << 3); // guard, round, sticky
    sig_b = (sig_b << 3);

```

对阶：小阶向大阶看齐

尾数左移留出GRS bits



[nemu/src/cpu/fpu.c](#)

}

```

uint32_t internal_float_add(uint32_t b, uint32_t a) {
    FLOAT f, fa, fb;
    fa.val = a;
    fb.val = b;
    ...
    sig_a = (sig_a << 3); // guard, round, sticky
    sig_b = (sig_b << 3);

    uint32_t sticky = 0;
    while (shift > 0)
    {
        sticky = sticky | (sig_a & 0x1);
        sig_a = sig_a >> 1;
        sig_a |= sticky;
        shift--;
    }
    ...
}

```

对阶：小阶向大阶看齐

尾数左移留出GRS bits

尾数右移对阶，注意粘位的操作

[nemu/src/cpu/fpu.c](#)


```
uint32_t internal_float_add(uint32_t b, uint32_t a) {  
    FLOAT f, fa, fb;  
    fa.val = a;  
    fb.val = b;  
  
    ...  
    if (fa.sign) { sig_a *= -1; }  
    if (fb.sign) { sig_b *= -1; }  
  
    sig_res = sig_a + sig_b;  
  
    if (sign(sig_res)) {  
        f.sign = 1;  
        sig_res *= -1;  
    }  
    else { f.sign = 0; }  
  
    ...  
}
```

尾数相加

根据符号，尾数相加得到中间结果

[nemu/src/cpu/fpu.c](#)

整数运算得到中间结果

- 浮点数做加法（减法）的步骤

1. 对阶：小阶向大阶看齐

小阶增加至大阶，同时尾数右移，保证对应真值不变

2. 尾数相加（相减）

$$\begin{aligned} & 1.1 \times 2^{10} + 1.11 \times 2^{12} \\ & \quad \downarrow \\ & = 0.011 \times 2^{12} + 1.11 \times 2^{12} \\ & = \underline{10.001 \times 2^{12}} \end{aligned}$$

不符合IEEE 754标准
必须进行规格化

浮点数的运算模拟（基本流程）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```

规格化与舍入

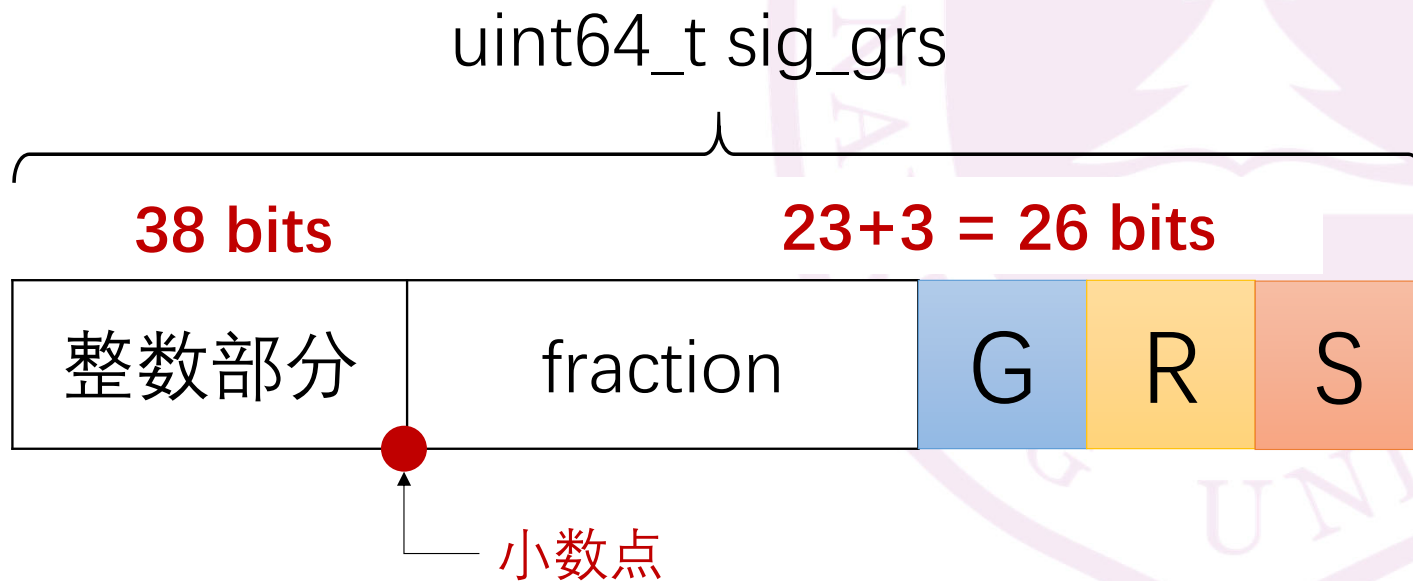
[nemu/src/cpu/fpu.c](#)

```
uint32_t internal_normalize(uint32_t sign,  
                           int32_t exp,  
                           uint64_t sig_grs  
                           )
```

规格化与舍入

nemu/src/cpu/fpu.c

```
uint32_t internal_normalize(uint32_t sign, // 结果的符号  
                           int32_t exp, // 中间结果阶数（含偏置常数，可能为负）  
                           uint64_t sig_grs // 中间结果尾数，26位小数  
                           )
```



规格化与舍入

对于加减法，中间结果 $\text{exp} \geq 0$

```
uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

Case 1: $\text{exp} > 0$ ，且， sig_grs 隐藏位后面超过了26位

条件： $\text{sig_grs} \gg 26 > 1$ 且 $\text{exp} > 0$

操作：将尾数右移1位， $\text{exp}++$ ，直至 $\text{sig_grs} \gg 26 == 1$

注意sticky bit的操作

例外： exp 加过了头（ $\geq 0xFF$ 了），阶码上溢

规格化与舍入

对于加减法，中间结果 $\text{exp} \geq 0$

```
uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

Case 2: $\text{exp} > 0$ ，且， sig_grs 隐藏位后面不足26位，如： $1.x + (-1.0)$

条件： $\text{sig_grs} \gg (23 + 3) == 0$ 且 $\text{exp} > 0$

操作：尾数左移1位， $\text{exp}--$ ，直至 $\text{sig_grs} \gg 26 == 1$

例外： exp 减过了头（ $==0$ 了），得到了非规格化数

注意为了配合非规格化数的阶码为0表示 2^{-126} ，需要额外将尾数右移一次，注意sticky bit的操作

规格化与舍入

对于加减法，中间结果 $\text{exp} \geq 0$

```
uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

Case 3: $\text{exp} == 0$ ，且， $\text{sig_grs} \gg 26 == 1$
需要将 $\text{exp}++$ ，保证阶码真值为-126

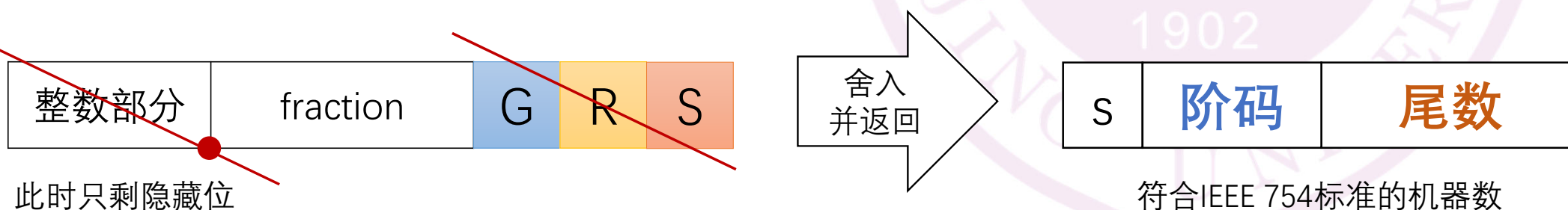
其它情形：无需进行规格化（有哪些情形？）
理解教程中的伪代码

规格化与舍入

```
uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

如果前面的过程没有产生溢出，根据GRS bits的取值情况进行舍入

- 就近舍入到偶数
- 舍入若产生尾数加1，有可能出现破坏规格化的情况
 - 此时需要进行额外的一次右规并判断阶码上溢的情况



浮点数的运算模拟（基本流程）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```

浮点数的运算模拟（基本流程）

```
uint32_t internal_float_add(uint32_t b, uint32_t a)
{
    1. 处理边界情况 (NaN、0、INF)
    2. 提取符号、阶码、尾数
    3. 整数运算得到中间结果
    4. 舍入并规格化后返回
}
```

乘法：尾数相乘，阶码相加
除法：尾数相除，阶码相减

注意偏置常数的加减操作

乘除法提示

- 乘法：尾数相乘，阶码相加

nemu/src/cpu/fpu.c

```
uint32_t internal_float_mul(uint32_t b, uint32_t a) {  
    ...  
    uint64_t sig_a, sig_b, sig_res;  
    sig_res = sig_a * sig_b;  
    uint32_t exp_res = 0;  
  
    /* TODO: exp_res = ? leave space for GRS bits. */  
    printf("\e[0;31mPlease implement me at fpu.c\e[0m\n");  
    assert(0);  
    ...  
}
```

乘除法提示

- 乘法：尾数相乘，阶码相加

[nemu/src/cpu/fpu.c](#)

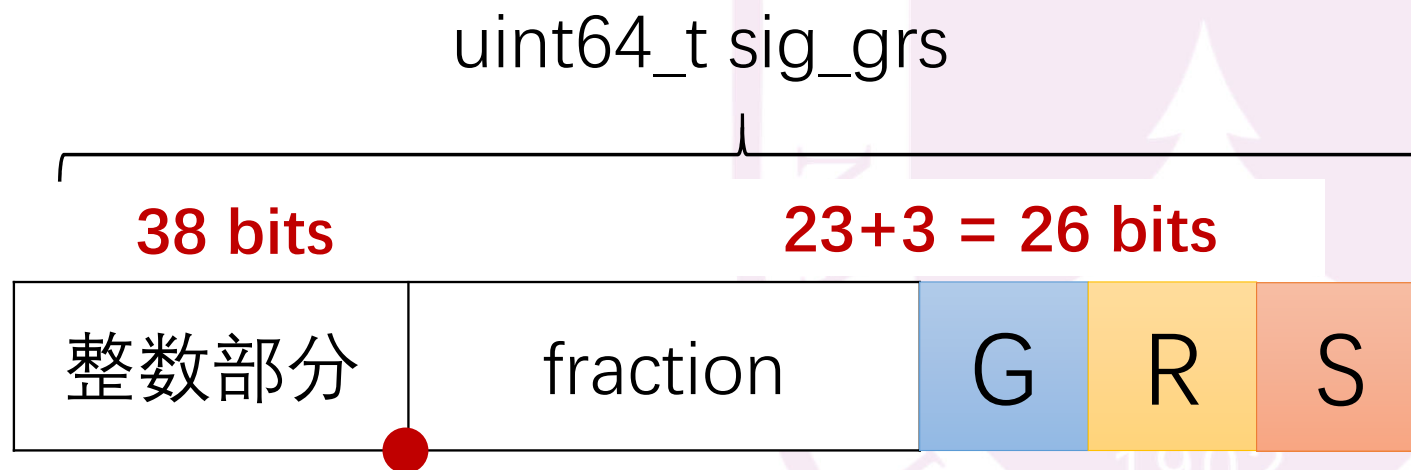


无符号整数相乘得到的中间结果，如何
对应带小数点的尾数真值相乘中间结果？

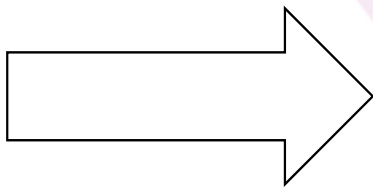
乘除法提示

- 乘法：尾数相乘，阶码相加

[nemu/src/cpu/fpu.c](#)



与我们约定的sig_grs的标准之间有没有不一致？

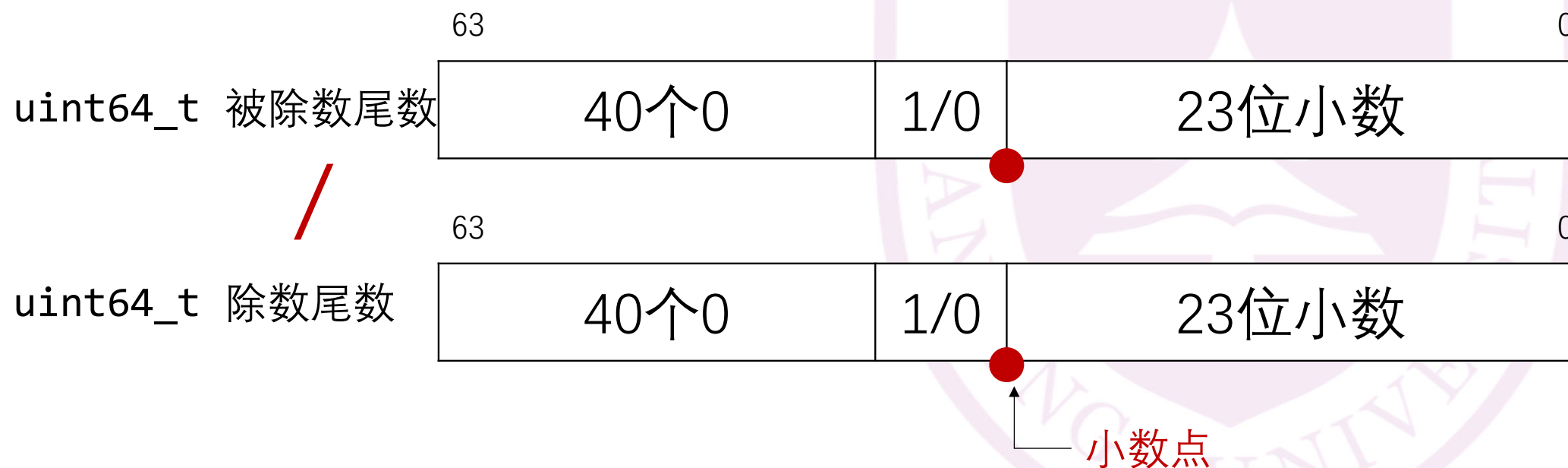


保持尾数中间结果编码不变，
通过调整阶码exp，使得真
值符合sig_grs约定

乘除法提示

- 除法：尾数相除，阶码相减

[nemu/src/cpu/fpu.c](#)



乘除法提示

- 除法：尾数相除，阶码相减

[nemu/src/cpu/fpu.c](#)

<< 左移直至高位没有0

63			0
uint64_t 被除数尾数	40个0	1/0	23位小数

/

63			0
uint64_t 除数尾数	40个0	1/0	23位小数

为提高计算精度所做的操作

同样需要调整阶码exp，
使得真值符合sig_grs约定

>> 右移直至低位没有0

乘除法的尾数规格化

乘除法这一条不再成立

~~对于加减法，中间结果 $\text{exp} \geq 0$~~

```
uint32_t internal_normalize(uint32_t sign, int32_t exp, uint64_t sig_grs)
```

加减法规则基础上的额外情形： $\text{exp} < 0$

操作：和 $\text{sig_grs} \gg 26 > 1$ 的情形一样，需要右规，直至

- 得到非规格化数： $\text{exp} == 0$ 且 $\text{sig_grs} \gg 26 \leq 1$ 且 $\text{sig_grs} > 0$ （舍入之后仍大于0）
 - 在while循环外多右移一次配合非规格化数阶码的约定
- 或，得到规格化数： $\text{exp} > 0$ 且 $\text{sig_grs} \gg 26 == 1$

例外：

已经无法右规了 $\text{sig_grs} \leq 4$ （舍入后就是0了）， exp 仍然小于0，产生阶码下溢

实验过程及要求

1. 实现 `nemu/src/cpu/fpu.c` 中的四个浮点数运算函数；
2. 将 `internal_normalize()` 函数补完；
3. 使用 `make` 命令编译项目；
4. 使用 `./nemu/nemu --test-fpu xxx` 或 `make test_pa-1` 命令执行 NEMU 并通过各个浮点数运算测试用例。

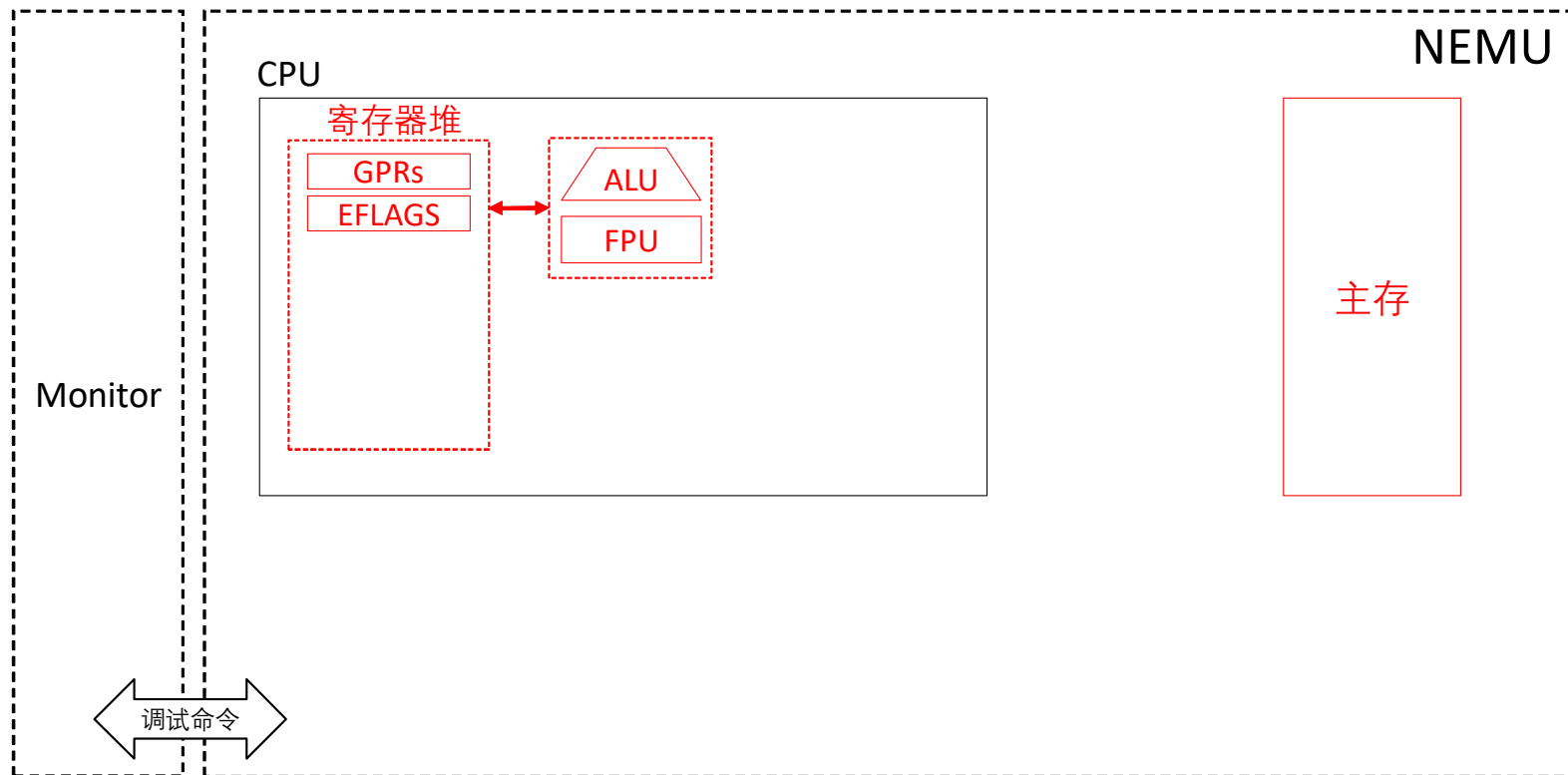
~PA 1-3顺利完成~

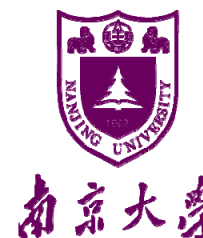
```
fpu_test_add() pass  
fpu_test_sub() pass  
fpu_test_mul() pass  
fpu_test_div() pass
```

在实验报告中，回答以下问题：

为浮点数加法和乘法各找两个例子：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；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。是否都能找到？若找不到，说出理由。

路线图进展





PA 1-3 结束

- 整个PA 1截止
 - 2020年10月8日（周四） 24:00
- 提交方式
 - `make submit_pa-1`
 - 下载submit/下面产生的压缩包
 - 将压缩包和实验报告提交到cslab中对应的备用窗口
 - 实验报告中需回答PA 1-1 到 PA 1-3教程后面的问题