华中农业大学 HUAZHONG AGRICULTURAL UNIVERSITY

Ch3: Arithmetic and Logic Operate and ALU
第三章 运算方法和运算部件

浮点运算和浮点运算器

- 指令集中与浮点运算相关的指令（以MIPS为例）
  - 涉及到的操作数
    - 单精度浮点数
    - 双精度浮点数
  - 涉及到的运算
    - 算术运算：加 / 减 / 乘 / 除
- 浮点数加减运算
- 浮点数乘除运算
- 浮点数运算的精度问题

回顾：浮点数的表示

浮点数真值：$S = \pm\ 2^{E} \times M$

浮点数机器表示：

Ef	E1	···	Em	Sf	M1	···	Mn
阶符	阶码			数符	尾数		

E：阶码，为定点整数，补码或移码表示。
其位数决定数值范围；阶符表示数的大小。

M：尾数，为定点小数，原码或补码表示。
其位数决定数的精度；数符表示数的正负。

MIPS中的浮点算术运算指令

FP add single	add.s	\$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6
FP subtract single	sub.s	\$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6
FP multiply single	mul.s	\$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6
FP divide single	div.s	\$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6
FP add double	add.d	\$f2,\$f4,\$f6	\$f2 = \$f4 + \$f6
FP subtract double	sub.d	\$f2,\$f4,\$f6	\$f2 = \$f4 - \$f6
FP multiply double	mul.d	\$f2,\$f4,\$f6	\$f2 = \$f4 × \$f6
FP divide double	div.d	\$f2,\$f4,\$f6	\$f2 = \$f4 / \$f6

MIPS提供专门的浮点数寄存器：

- 32个32位单精度浮点数寄存器：\$f0, \$f1, ……, \$f31
- 连续两个寄存器存放一个双精度浮点数

浮点操作数：32位单精度 / 64位双精度浮点数

浮点操作：加 / 减 / 乘 / 除

MIPS中的浮点数传送指令

load word copr. 1	lwc1 $f1,100($s2)	$f1 = Memory[$s2 + 100]
store word copr. 1	swc1 $f1,100($s2)	Memory[$s2 + 100] = $f1

涉及到的浮点操作：传送操作（与定点传送一样）

涉及到的浮点操作数：32位单精度浮点数

MIPS中的浮点数比较和分支指令

branch on FP true	bc1t 25	if (cond == 1) go to PC + 4 + 100
branch on FP false	bc1f 25	if (cond == 0) go to PC + 4 + 100
FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0
FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond = 1; else cond = 0

涉及到的浮点操作：比较操作（用减法来实现比较）

涉及到的浮点操作数： 32位单精度浮点数/64位双精度浮点数

MIPS浮点运算指令的总结

- 浮点操作数的表示
  - 32位单精度浮点数 / 64位双精度浮点数
- 浮点数的运算
  - 加法 / 减法 / 乘法 / 除法

IA-32中浮点数寄存器是80位，这可能会给float和double类型变量的运算带来隐患。

例子：将以下程序编译为MIPS汇编语言

```
Float f2c (float fahr)
{
  return ((5.0 / 9.0) * (fahr-32.0));
}
```

假设变量fahr存放在$f12中，返回结果存放在$f0中。三个常数存放在通过$gp能访问到的存储单元中。

```
f2c : lwc1   $f16, const5($gp)
      lwc1   $f18, const9($gp)
      div.s  $f16, $f16, $f18
      lwc1   $f18, const32($gp)
      sub.s  $f12, $f12, $f18
      mul.s  $f0, $f16, $f12
      jr  $ra
```

有关Floating-point number的问题

实现一套浮点数运算指令，要解决的问题有：

Issues:

- Representation(表示)：
  Normalized form （规格化形式） 和 Denormalized form
  单精度格式 和 双精度格式
- Range and Precision （表数范围和精度）
- Arithmetic （+, -, *, / ）
- Rounding(舍入)
- Exceptions (e.g., divide by zero, overflow, underflow)
  （异常处理：如除数为0，上溢，下溢等）
- Errors （误差）与精度控制

浮点数运算及结果

设两个规格化浮点数分别为 $A=M_a \cdot 2^{Ea}$ $B=M_b \cdot 2^{Eb}$ ，则：

$A\pm B=(M_a \pm M_b \cdot 2^{-(Ea-Eb)}) \cdot 2^{Ea}$ （假设Ea>=Eb ） **1.5+1.5=?**

$A*B=(M_a * M_b) \cdot 2^{Ea+Eb}$ **1.5-1.0=?**

$A/B=(M_a / M_b) \cdot 2^{Ea-Eb}$

上述运算结果可能出现以下几种情况： **SP最大指数为多少？ 127！**

阶码上溢：一个正指数超过了最大允许值 => +∞/-∞/溢出

阶码下溢：一个负指数超过了最小允许值 => +0/-0 **SP最小指数为多少？ -126！**

尾数溢出：最高有效位有进位 => 右规 **尾数溢出，结果不一定溢出**

非规格化尾数：数值部分高位为0 => 左规 **运算过程中添加保护位**

右规或对阶时，右段有效位丢失 => 尾数舍入

IEEE建议实现时为每种异常情况提供一个自陷允许位。若某异常对应的位为1，则发生相应异常时，就调用一个特定的异常处理程序执行。

IEEE754标准规定的五种异常情况

① 无效运算（无意义）
- 运算时有一个数是非有限数，如：
  加 / 减∞、0 x ∞、 ∞/∞等
- 结果无效，如：
  源操作数是NaN、0/0、x REM 0、 ∞ REM y 等

② 除以0（即：无穷大）
③ 数太大（阶码上溢）：对于SP，指阶码 E >1111 1110 （指数>127）
④ 数太小（阶码下溢） ：对于SP，指阶码 E < 0000 0001（指数<-126 ）
⑤ 结果不精确（舍入时引起），例如1/3，1/10等不能精确表示成浮点数

浮点数加/减运算

- 十进制科学计数法的加法例子

  $0.123 \times 10^5 + 0.560 \times 10^2$

  其计算过程为：

  $0.123 \times 10^5 + 0.560 \times 10^2 = 0.123 \times 10^5 + 0.000560 \times 10^5$

  $= (0.123 + 0.00056) \times 10^5 = 0.12\boxed{356} \times 10^5$

  $= 0.124 \times 10^5$

  进行尾数加减运算前，必须“对阶”！最后还要考虑舍入

  计算机内部的二进制运算也一样！

- “对阶”操作：目的是使两数阶码相等
  - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
  - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

对阶

- **如何对阶？**

计算$[\Delta E]_{补}$判断指数大小，小阶向大阶看齐，阶小的尾数右移，移出的低位保留到“附加位”上。

$[\Delta E]_{补} = [Ex-Ey]_{补} = [Ex]_{移} + [-[Ey]_{移}]_{补} \pmod{2^n}$

提问：IEEE754 SP中，|ΔE|大于多少时，结果就等于阶大的那个数(即小数被大数吃掉)?

24！

1.xx…x → 0. 00…01xx…x(右移24位后，尾数变为0)

提问：偏置常数是127，会不会影响阶码运算？

对计算$[Ex-Ey]_{补} \pmod{2^n}$ **没有影响**

$[\Delta E]_{补} = 256+Ex-Ey = 256+127+Ex-(127+Ey)$

$= 256 + [Ex]_{移} - [Ey]_{移} = [Ex]_{移} + [-[Ey]_{移}]_{补} \pmod{256}$

IEEE 754 浮点数加法运算举例

已知float x=0.5， y=-0.4375， 求x+y=?

解： $x=0.5=1/2=(0.100...0)_2=(1.00...0)_2 \times 2^{-1}$

$y=-0.4325=(-0.01110...0)_2=(-1.110..0)_2 \times 2^{-2}$

$[x]_{浮}$=0 01111110,00…0 $[y]_{浮}$=1 01111101,110…0

对阶：$[\Delta E]_{补}$=0111 1110 + 1000 0011=0000 0001， ΔE=1

故y需对阶：$[y]_{浮}$=1 0111 1110， 1110…0(高位补隐藏位)

尾数相加：01.0000...0 + (10.1110...0) = 00.00100…0

(原码加法，符号位分开处理)

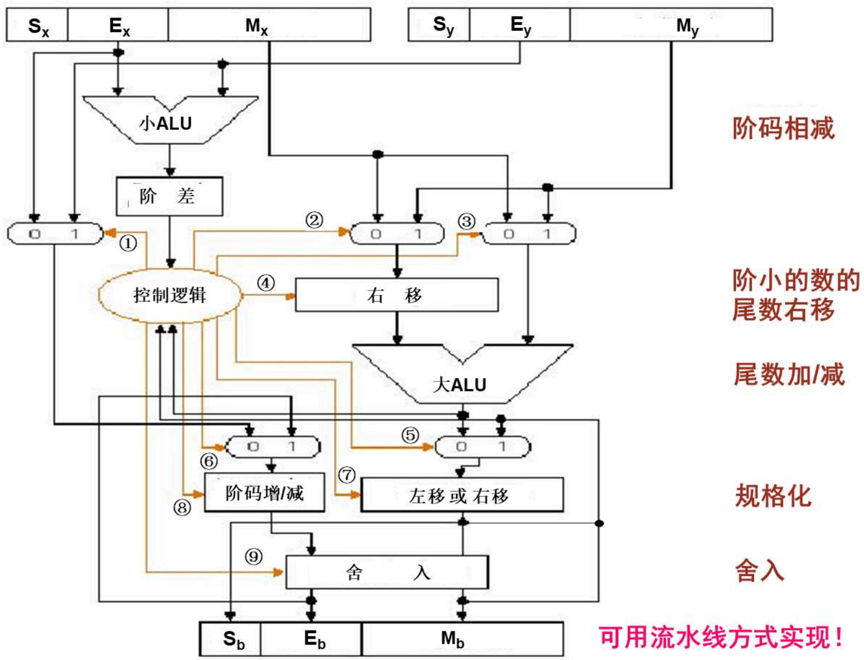
左规：尾数左移3位，阶码减3

$+(0.00100\cdots0)_2 \times 2^{-1} = +(1.00\cdots0)_2 \times 2^{-4}$

$[x+y]_{浮}$=0 0111 1011 00…0=0331 8000H

$x+y=(1.0)_2 \times 2^{-4}=1/16=0.0625$

无舍入，无溢出。



可用流水线方式实现！

例. 一维向量x, y的求和表达式如下, 试用4段的浮点加法流水线来实现一维向量的求和运算. 这四段流水线是阶码比较, 对阶, 尾数相加, 规格化. 要求画出向量加法计算流水时空图.

$$\begin{pmatrix} 56 \\ 20.5 \\ 0 \\ 114.3 \\ 69.6 \\ 3.14 \end{pmatrix} + \begin{pmatrix} 65 \\ 14.6 \\ 336 \\ 7.2 \\ 72.8 \\ 1.41 \end{pmatrix} = \begin{pmatrix} 121 \\ 35.1 \\ 336 \\ 121.5 \\ 142.4 \\ 4.55 \end{pmatrix}$$

2、浮点数的乘除法运算

- **浮点数乘法：**$X*Y = (X_m * Y_m) \cdot 2^{Xe+Ye}$
- **浮点数除法：**$X/Y = (X_m / Y_m) \cdot 2^{Xe-Ye}$

浮点数尾数采用原码乘/除运算。

浮点数乘 / 除法步骤

(1) 求阶： $Xe \pm Ye \mp 127$ （见下页）

(2) 尾数相乘除： Xm */Ym （原码乘/除）

(3) 两数符号相同结果为正；相异为负；

(4) 结果的尾数高位为0需左规；最高位有进位需右规。

(5) 若尾数比规定的长，则需舍入。

(6) 若尾数是0，则阶码也置0，结果为0。

(7) 阶码溢出判断

浮点数的乘除法和加减法，区别仅在于：

指数的加减，尾数的乘除。

求阶码的和、差

设Ex和Ey是两个阶码，则其和差计算方法为：

- 阶码加法公式为： $E_{和} \leftarrow Ex+Ey+129 \pmod{2^8}$

$[Ex+Ey]_{移}$ = 127 + Ex+ Ey = 127 + Ex + 127 + Ey − 127

$E_{和}$ = $[Ex]_{移} + [Ey]_{移} - 127$

= $[Ex]_{移} + [Ey]_{移} + [-127]_{补}$

= $[Ex]_{移} + [Ey]_{移} + 10000001B \pmod{2^8}$

- 阶码减法公式为： $E_{差} \leftarrow Ex+[-Ey]_{补}+127 \pmod{2^8}$

$[Ex - Ey]_{移}$ = 127 + Ex − Ey = 127+Ex − (127+Ey)+127

$E_{差}$ = $[Ex]_{移} - [Ey]_{移} + 127$

= $[Ex]_{移} + [-[Ey]_{移}]_{补} + 01111111B \pmod{2^8}$

阶码和差计算举例

设Ex和Ey是两个移码，求是和/差的移码Eb。

例：两个阶码的真值为10和-5，求10+(-5)和10-(-5)的移码。

解：Ex = 127+10 =137=1000 1001B

Ey = 127+ (–5) = 122 = 0111 1010B

$[–Ey]_补$= 1000 0110B

将Ex和Ey代入上页公式，得：

Eb = Ex+Ey +129 = 1000 1001 + 0111 1010 +1000 0001

= 1000 0100B = 132 （mod 2^8）

和为132 – 127 = 5，正好等于10 + (–5) = 5。

Eb = Ex+$[–Ey]_补$+127 = 1000 1001 + 1000 0110 +0111 1111

= 1000 1110B = 142 （mod 2^8）

差为142–127 = 15，正好等于10 – (–5) = 15。

Extra Bits(附加位)

IEEE754规定：中间结果在右边加至少2个附加位 (guard & round)

Guard bit(保护位)：在significand右边的位

Rounding bit(舍入位)：在保护位右边的位

作用：保护对阶时右移的位和中间结果。

左规时移到尾数中；或作为舍入的依据。

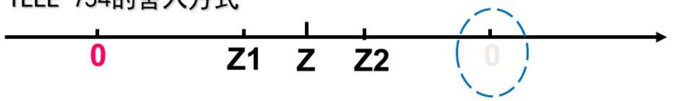
加更多附加位一般可得到更高的精度。

Add/Sub:

	1.xxxxx	1.xxxxx	1.xxxxx	1.xxxxxxxx
+	1.xxxxx	0.001xxxxx	0.01xxxxx	-1.xxxxxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy	0.0…0xxxx

IEEE 754中附加位的舍入方式

IEEE 754的舍入方式



(Z1和Z2是Z可表示的最近的左、右两个数)

(1) 朝+∞方向舍入:舍入为Z2

(2) 朝-∞方向舍入:舍入为Z1

(3) 朝0方向舍入：截去。正数：取Z1； 负数：取Z2

(4) 就近舍入(default)：舍入为最近可表示的数

01：舍
11：入
10：(强迫结果为偶：最低位为0则舍掉多余位，最低位为1则进位1)

例：1.110101 → 1.1101； 1.110111 → 1.1110；
1.111010 → 1.1110； 1.110110 → 1.1110；

IEEE 754的舍入方式说明

IEEE 754建议可通过在舍入位后再引入粘位 “ sticky bit”增强精度

加减运算对阶过程中，若阶码较小的数的尾数右移时，舍入位之后有非0数，则可设置sticky bit。

举例：

$1.24 \times 10^4 + 5.03 \times 10^1$ 分别采用一位、二位、三位附加位时，结果各是多少？（就近舍入到偶数）

尾数精确结果为1.24503, 所以分别为：

1.24，1.24，1.25

浮点数的溢出判断

以下情况下，可能会导致阶码溢出

- 左规（阶码减 1可能下溢）
  - 左规时：先判断阶码是否为全0，若是，则直接置阶码下溢；否则，阶码减1后判断阶码是否为全0，若是，则阶码下溢。
- 右规（阶码加1可能上溢）
  - 右规时，先判断阶码是否为全1，若是，则直接置阶码上溢；否则，阶码加1后判断阶码是否为全1，若是，则阶码上溢。

问题：机器内部如何减1？ $+[-1]_{补} = +11\ldots1$

浮点运算器实例（自学）

- 80×87协处理器
- 奔腾CPU 的流水线浮点运算部件

小组讨论1

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
        1234567936.000000
        1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

小组讨论2-非规格化浮点数

当结果为 0.1×2^{-126} 时，是用非规格化数表示还是近似为0?

以下程序试图计算 $2^{-63}/2^{64}=2^{-127}$

```
#include <stdio.h>

main()
{ float x=1.084202172485504e-19;
  float y=1.844674407370955e+19;
  float z=x/y;
  printf("x=%f %x\n",x,x);
  printf("y=%f %x\n",y,y);
  printf("z=%f %x\n",z,z);
}
```

计算器

2-63

2^{64}

```
user@debian:~/Templates$ ./denom
x=0.000000 0
y=18446744073709551616.000000 0
z=0.000000 0
```

讨论问题：

1. **计算器上算的准确吗?**
2. **为什么 x 输出为 0?**
3. **为什么 y 的输出发生变化?**
4. **为什么x、y、z用%x输出为0?**
5. **Z 输出为 0 说明了什么?**
6. **如下赋初值对否?**

   float x=0x40000000;

   float y=0x5f800000;

小组讨论3

```c
#include <stdio.h>

main()
{ float x=0.0000152587890625;  2-16
  float y=0.000030517578125;  2-15
  float z=x*x*x*y;
  float l=1.8446744073709551616e+19;
  float m=z/l;
  printf("z=%e \n",z);
  printf("m=%1.38e \n",m);
  //printf("z=%f %x\n",z,z);
}
```

```
user@debian:~/Templates$ gcc -o denom denom.c
user@debian:~/Templates$ ./denom
z=1.084202e-19
m=5.87747175411143753984368268611122838909e-39
```

当结果为 0.1×2^{-126} 时，用非规格化数表示，而不是近似表示成0！

z和m的输出结果说明了什么?

小 结

- 浮点运算指令（ 以MIPS为参考 ）
- 浮点数的表示（IEEE754标准）
  - 单精度SP（float）和双精度DP（double）
    - 规格化数(SP)：阶码1~254，尾数最高位隐含为1
    - 0(阶为全0，尾为全0)
    - ∞(阶为全1，尾为全0)
    - NaN(阶为全0，尾为非0)
    - 非规数(阶为全1，尾为非0)
- 浮点数加减运算
  - 对阶、尾数加减、规格化（上溢/下溢处理）、舍入
- 浮点数乘除运算
  - 求阶、尾数乘除、规格化（上溢/下溢处理） 、舍入
- 浮点数的精度问题
  - 中间结果加保护位、舍入位（和粘位）
  - 最终进行舍入（有四种舍入方式）
    - 就近（中间值强迫为偶数）、+ ∞方向、- ∞方向、0方向
    - 默认为“就近”舍入方式

本章总结（1）

- ALU的实现
  - 算术逻辑单元ALU：实现基本的加减运算和逻辑运算。
  - 加法运算是所有定点和浮点运算（加/减/乘/除）的基础，加法速度至关重要
  - 进位方式是影响加法速度的重要因素
  - 并行进位方式能加快加法速度
  - 通过“进位生成”和“进位传递”函数来使各进位独立、并行产生

本章总结（2）

定点数运算：由ALU + 移位器实现各种定点运算

- 移位运算
  - 逻辑移位：对无符号数进行，左（右）边补0，低（高）位移出
  - 算术移位：对带符号整数进行，移位前后符号位不变，编码不同，方式不同。
- 扩展运算
  - 零扩展：对无符号整数进行高位补0
  - 符号扩展：对补码整数在高位直接补符
- 加减运算
  - 补码加/减运算：用于整数加/减运算。符号位和数值位一起运算，减法用加法实现。同号相加时，若结果的符号不同于加数的符号，则会发生溢出。
  - 原码加/减运算：用于浮点数尾数加/减运算。符号位和数值位分开运算，同号相加，异号相减；加法直接加；减法用加负数补码实现。
- 乘法运算：用加法和右移实现。
  - 补码乘法：用于整数乘法运算。符号位和数值位一起运算。采用Booth算法。
  - 原码乘法：用于浮点数尾数乘法运算。符号位和数值位分开运算。数值部分用无符号数乘法实现。
- 除法运算：用加/减法和左移实现。
  - 补码除法：用于整数除法运算。符号位和数值位一起运算。
  - 原码除法：用于浮点数尾数除法运算。符号位和数值位分开运算。数值部分用无符号数除法实现。

本章总结（3）

- 浮点数运算：由多个ALU + 移位器实现
  - 加减运算
    - 对阶 、尾数相加减、规格化处理、舍入、判断溢出
  - 乘除运算
    - 尾数用定点原码乘/除运算实现，阶码用定点数加/减运算实现。
  - 溢出判断
    - 当结果发生阶码上溢时，结果发生溢出，发生阶码下溢时，结果为0。
  - 精确表示运算结果
    - 中间结果增设保护位、舍入位
    - 最终结果舍入方式：就近舍入 / 正向舍入 / 负向舍入 / 截去。

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
1234567936.000000
1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

float可精确表示7个十进制有效数位，后面的数位是舍入后的结果，舍入后的值可能会更大，也可能更小

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

当结果为 0.1×2^{-126} 时，是用非规格化数表示还是近似为0?

以下程序试图计算 $2^{-63}/2^{64}=2^{-127}$

```
#include <stdio.h>

main()
{ float x=1.084202172485504e-19;
  float y=1.844674407370955e+19;
  float z=x/y;
  printf("x=%f %x\n",x,x);
  printf("y=%f %x\n",y,y);
  printf("z=%f %x\n",z,z);
}
```

计算器 2^{-63} 2^{64}

```
user@debian:~/Templates$ ./denom
x=0.000000 0
y=18446744073709551616.000000 0
z=0.000000 0
```

讨论问题：

1. 计算器上算的准确吗?
2. 为什么 x 输出为 0?
3. 为什么 y 的输出发生变化?
4. 为什么x、y、z用%x输出为0?
5. Z 输出为 0 说明了什么?
6. 如下赋初值对否?
   float x=0x40000000;
   float y=0x5f800000;

```
#include <stdio.h>

main()
{ float x=0.0000152587890625;
  float y=0.000030517578125;
  float z=x*x*x*y;
  float l=1.8446744073709551616e+19;
  float m=z/l;
  printf("z=%e \n",z);
  printf("m=%1.38e \n",m);
  //printf("z=%f %x\n",z,z);
}
```

2^{-16} 2^{-15}

```
user@debian:~/Templates$ gcc -o denom denom.c
user@debian:~/Templates$ ./denom
z=1.084202e-19
m=5.87747175411143753984368268611122838909e-39
```

当结果为 0.1×2^{-126} 时，用非规格化数表示，而不是近似表示成0！

z和m的输出结果说明了什么?