数据的机器级表示与运算

浮点数运算

主讲人：邓倩妮

上海交通大学

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

浮点数的算术精确性

- $x +_f y = \text{Round}(x + y)$

- $x \times_f y = \text{Round}(x \times y)$

- 基本思想：
- 首先进行计算，
- 然后将计算结果表示为规定的浮点数标准格式
  - 如果阶（exponent）太大，就溢出
  - 将运算结果的尾数位舍入到 **`frac`** 规定的长度

四种舍入模式

	$1.40	$1.60	$1.50	$2.50	–$1.50
向0（截断）	$1	$1	$1	$2	–$1
向正无穷大（向上）	$2	$2	$2	$3	–$1
向负无穷大（向下）	$1	$1	$1	$2	–$2
首选“偶数”值(默认)	$1	$2	$2	$2	–$2

- 默认模式：最近舍入（Round to Nearest），首选“偶数”值
- 与四舍五入相比：不是.5的舍入，同四舍五入
- .5的舍入，采用取偶数的方式。例如：
  - 最近舍入模式：Round(0.5) = 0; Round(1.5) = 2; Round(2.5) = 2;
  - 四舍五入模式：Round(0.5) = 1; Round(1.5) = 2; Round(2.5) = 3;

首选“偶数”值 Round-To-Even

- 当数值处于中间时,怎么做？
  - 一半时间向上舍入，一半时间向下舍入
- 减少运算中产生的误差
  - 例如：多个正数的加法，will consistently be over- or under- estimated
- Java支持的模式

- 在其他位置上的舍入，规则相同
- 当数值处于中间时, Round so that least significant digit is even
  - E.g., round to nearest hundredth

7.89<u>49999</u>	7.89	(Less than half way)
7.89<u>50001</u>	7.90	(Greater than half way)
7.89<u>50000</u>	7.90	(Half way—round up)
7.88<u>50000</u>	7.88	(Half way—round down)

二进制数的最近舍入

- 二进制数 的 Round-To-Even
  - 偶数 “Even” ：最低有效位为 0
  - 数值处于中间 “Half way” 要舍弃的位数的形式 $= 100\cdots_2$

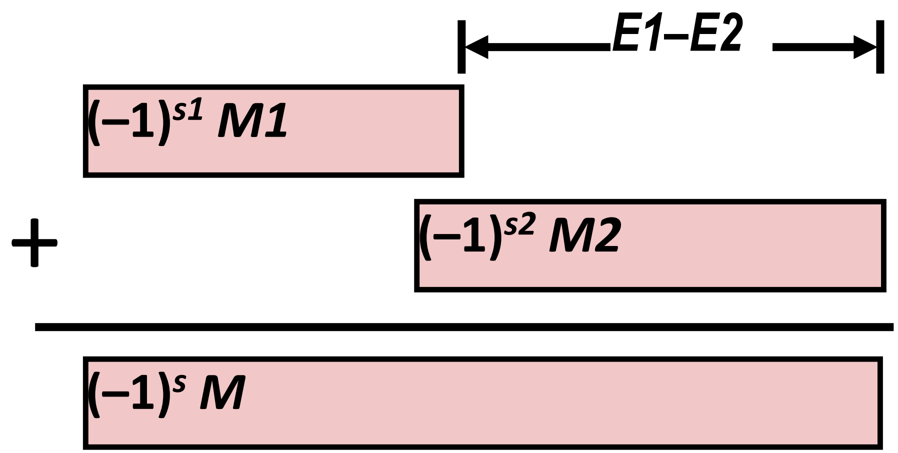
- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	$10.00\underline{011}_2$	10.00_2	(<1/2—down)	2
2 3/16	$10.00\underline{110}_2$	10.01_2	(>1/2—up)	2 1/4
2 7/8	$10.11\underline{100}_2$	11.00_2	(1/2—up)	3
2 5/8	$10.10\underline{100}_2$	10.10_2	(1/2—down)	2 1/2

浮点数加法

- $(-1)^{s1}\, \boldsymbol{M1}\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\, \boldsymbol{M2}\ \ 2^{E2}$
  - 假定 $E1 > E2$

Get binary points lined up



- 运算步骤：
  - 对阶
  - 尾数加减
  - 规格化（左规，右规）
  - 舍入
  - 检查溢出

- 运算结果: $(-1)^{s}\, \boldsymbol{M}\ \ 2^{E}$

Exercise

- Given A=2.6125×10^{1}, B=4.150390625×10^{-1}, Calculate the sum of A and B by hand, assuming A and B are stored by the following format, Assume 1 guard(保护位）, 1 round bit（舍入位）, and 1 sticky bit（粘贴位）, and round to the nearest even（首选“偶数”值舍入 ）. Show all the steps.

- Solution:

$2.6125\times10^1 + 4.150390625\times10^{-1}$

$2.6125\times10^1 = 26.125 = 11010.001 = 1.1010001000\times2^4$

$4.150390625\times10^{-1} = .4150390625 = .011010100111$

$=1.1010100111\times2^{-2}$ （对阶，小阶往大阶对）

Shift binary point 6 to the left to align exponents,

GR

1.1010001000 00

+.0000011010 10 0111 (Guard = 1, Round = 0, Sticky = 1)

\-

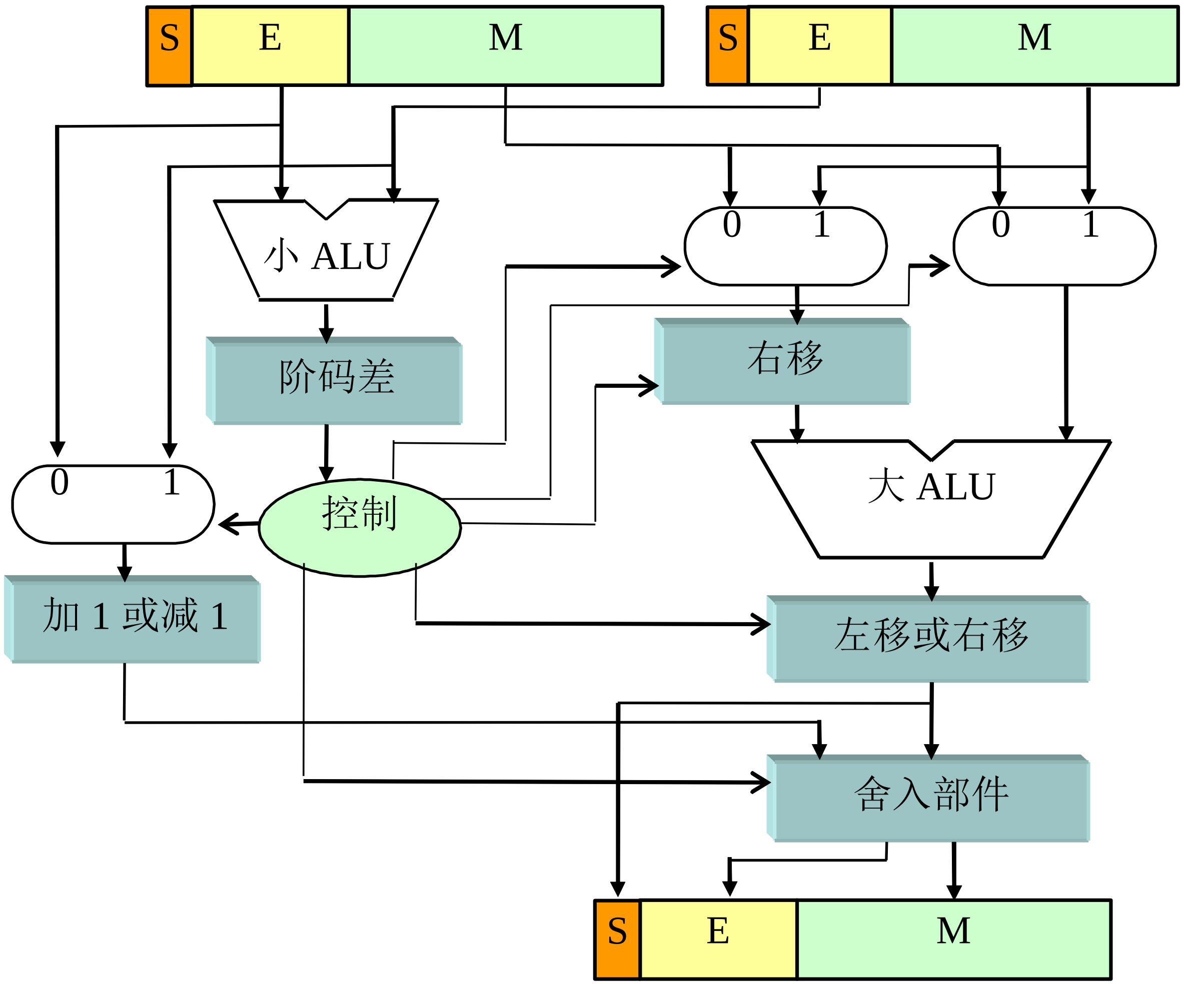
1.1010100010 10 （尾数相加） and （尾数规格化检查）

the extra bits (G,R,S) are more than half of the least significant bit (0). Thus, the value is rounded up. （舍入）

1.1010100011×2^4（检查，无溢出）

$= 11010.100011 \times 2^0 = 26.546875 = 2.6546875 \times 10^1$

浮点加法运算电路



FP Add 的算术特性

- 与整数加法（形成的阿贝尔群）比较
  - 加法封闭 Closed under addition? **Yes**
    - But may generate infinity or NaN
  - 交换性 Commutative? **Yes**
  - 结合性 Associative? **No**
    - 溢出、 舍入导致的不精确
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 是加法单位元 additive identity? **Yes**
  - 每个元素都有加法逆元（ additive inverse ）？ **Almost**
    - Yes, except for infinities & NaNs
- 单调性 Monotonicity **Almost**
  - a ⩾ b ⇒ a+c ⩾ b+c?
    - Except for infinities & NaNs

浮点乘法 FP Multiplication

- $(-1)^{s1}\ \boldsymbol{M1}\ \ 2^{\boldsymbol{E1}}\ \ \times\ \ (-1)^{\boldsymbol{s2}}\ \boldsymbol{M2}\ \ 2^{\boldsymbol{E2}}$
- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ \ 2^{\boldsymbol{E}}$
- 步骤：
  - 阶码加
  - 尾数乘
  - 规格化
  - 舍入
  - 检查溢出
- Fixing
  - If $M \geqslant 2$, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit **`frac`** precision

FP Mult的算术特性

- 与整数乘法（可交换的环 Commutative Ring ）比较
  - 乘法封闭 Closed under multiplication? **Yes**
    - But may generate infinity or NaN
  - 交换性 Multiplication Commutative? **Yes**
  - 结合性 Multiplication is Associative? **No**
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20= inf`, `1e20*(1e20*1e-20)= 1e20`
  - 1 是乘法单位元 multiplicative identity? **Yes**
  - Multiplication distributes over addition? **No**
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)= 0.0`, `1e20*1e20 – 1e20*1e20 = NaN`
- 单调性 Monotonicity
  - $a \geqslant b$ & $c \geqslant 0 \Rightarrow a * c \geqslant b * c$? **Almost**
    - Except for infinities & NaNs

Floating Point Multiplication Example

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

Step 0: Hidden bits restored in the representation above

Step 1: Add the exponents (not in bias would be -1 + (-2) = -3 and in bias would be (-1+127) + (-2+127) – 127 = (-1 - 2) + (127+127-127) = -3 + 127 = 124

Step 2: Multiply the significands

1.0000 x 1.110 = 1.110000

Step 3: Normalized the product, checking for exp over/underflow

1.110000×2^{-3} is already normalized

Step 4: The product is already rounded, so we're done

Step 5: Rehide the hidden bit before storing

MIPS 浮点运算指令

- MIPS 有独立的浮点寄存器文件 (`$f0, $f1, …, $f31`) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them

```
lwcl  $f1,54($s2)     #$f1 = Memory[$s2+54]
swcl  $f1,58($s4)     #Memory[$s4+58] = $f1
```

- 支持 IEEE 754 单精度 single

```
add.s $f2,$f4,$f6     #$f2 = $f4 + $f6
```

和双精度运算 double precision operations

```
add.d $f2,$f4,$f6   #$f2||$f3 =
                          $f4||$f5 + $f6||$f7
```

similarly for `sub.s, sub.d, mul.s, mul.d, div.s, div.d`

MIPS 浮点运算指令续

- 单精度浮点数比较
- `c.x.s $f2,$f4`　　#if($f2 < $f4) cond=1; else cond=0

  where x may be `eq, neq, lt, le, gt, ge`
- 双精度浮点数比较

  `c.x.d $f2,$f4`　　#$f2||$f3 < $f4||$f5 cond=1; else cond=0
- 根据浮点比较结果转移 branch operations

  `bclt 25`　　#if(cond==1) go to PC+4+25

  `bclf 25`　　#if(cond==0) go to PC+4+25

Frequency of Common MIPS Instructions

- Only included those with >3% and >1%

	SPECint	SPECfp
addu	5.2%	3.5%
addiu	9.0%	7.2%
or	4.0%	1.2%
sll	4.4%	1.9%
lui	3.3%	0.5%
lw	18.6%	5.8%
sw	7.6%	2.0%
lbu	3.7%	0.1%
beq	8.6%	2.2%
bne	8.4%	1.4%
slt	9.9%	2.3%
slti	3.1%	0.3%
sltu	3.4%	0.8%

	SPECint	SPECfp
add.d	0.0%	10.6%
sub.d	0.0%	4.9%
mul.d	0.0%	15.0%
add.s	0.0%	1.5%
sub.s	0.0%	1.8%
mul.s	0.0%	2.4%
l.d	0.0%	17.5%
s.d	0.0%	4.9%
l.s	0.0%	4.2%
s.s	0.0%	1.1%
lhu	1.3%	0.0%

C语言中的浮点运算

- C 语言支持两种精度的浮点数 **float** & **double**
- 不同类型间的转换
  - Casting between **int**, **float**, and **double** changes bit representation
  - **double/float** → **int**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to Tmin(最小整数)
  - **int** → **double**
    - Exact conversion, as long as **int** has ≤ 53 bit word size
  - **int** → **float**
    - Will round according to rounding mode
    - 单精度：（有效尾数24位，相当于7位十进制有效位数）；int型有效位数31位，相当于10位十进制有效位

Discussion 1:

- What about following type converting: will it output true?

```
if ( i == (int) ((float) i) )  {
        printf (“true”);
}
if ( f == (float) ((int) f) )  {
        printf (“true”);
}
```

Question II about IEEE 754

- How about FP add associative?
- (X+Y)+Z=X+(Y+Z)

$x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$

$(x+y)+z = (-1.5\times10^{38}+1.5\times10^{38}) + 1.0 = 1.0$

$x+(y+z) = -1.5\times10^{38} + (1.5\times10^{38}+1.0) = 0.0$

Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0` ⇒ `((d*2) < 0.0)`
- `d > f` ⇒ `-f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

总结

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form M x 2^E
- One can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers

谢谢！