컴퓨터 공학 개론
Lecture 8

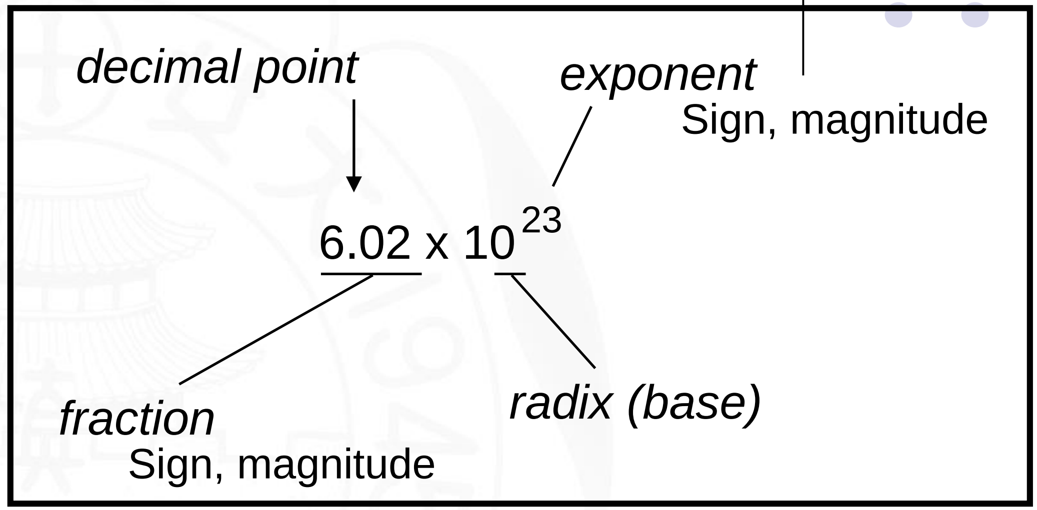
2017

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types **`float`** and **`double`** in C
  - vs. **`int`** and **`long`**

Recall Scientific Notation

- Number represented as
  - fraction
  - exponent (using radix 10)
- Arithmetic
  - Multiplication, Division
    - multiply/divide fraction
    - add/subtract exponent
    - normalize
    - example: $(5.6\text{x}10^{11}) \text{ x } (6.7\text{x}10^{12}) = (5.6\text{x}6.7)\text{x}10^{(11+12)}$
    - $= 37.52\text{x}10^{23} = 3.752\text{x}10^{24}$
  - Addition, Subtraction
    - convert operands to have the same exponent value
    - add/subtract fraction
    - normalize (if needed)
    - example: $(2.1\text{x}10^{3}) + (4.3\text{x}10^{4}) = (0.21\text{x}10^{4}) + (4.3\text{x}10^{4}) = 4.51\text{x}10^{4}$



Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits double: 11 bits		single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand (== (1+Fraction))
  - 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

- What number is represented by the single-precision float

  1 10000001 01000⋯00
  - S = 1
  - Exponent = 10000001_2 = 129
  - Fraction = $0.01000\cdots00_2$ ($= 0\text{x}2^{-1}+1\text{x}2^{-2}+0\text{x}2^{-3}+ \cdots$)
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\cdots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 − 127 = −126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 − 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 − 1023 = −1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 − 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Infinities and NaNs

- Exponent = 111…1, Fraction = 000…0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111…1, Fraction ≠ 000…0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0, sqrt(−4)
  - Can be used in subsequent calculations

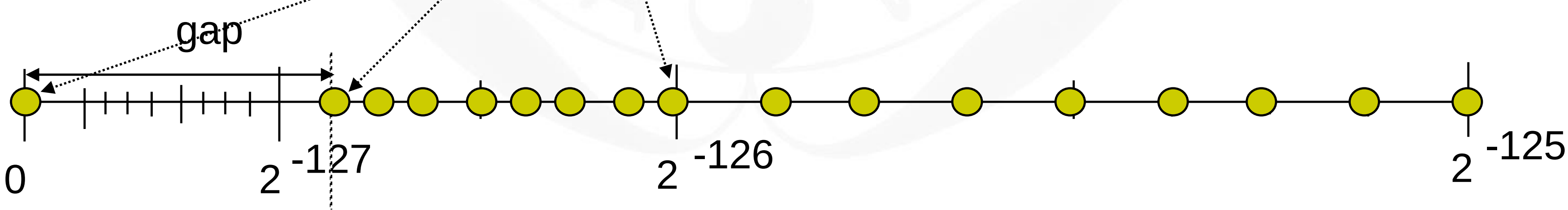
Range of Numbers Represented by IEEE 754

sign	Exponent	Fraction		
0/1	1111 1111	1111 1111 … 1111 1111	NaN	
0/1	1111 1111	…		
0/1	1111 1111	0000 0000 … 0000 0001		
0/1	1111 1111	0000 0000 … 0000 0000	± Infinity	
0/1	**1111 1110**	**1111 1111 … 1111 1111**	**$\pm$ 1.11..1 x $2^{254-127}$**	**~ $\pm$ 3.40 x 10^{38}**
0/1	**1111 1110**	**1111 1111 … 1111 1110**	**$\pm$ 1.11..0 x $2^{254-127}$**	
…	**…**	**…**	**…**	
0/1	**0000 0010**	**0000 0000 … 0000 0000**	**$\pm$ 1.00..0 x 2^{2-127}**	
0/1	**0000 0001**	**1111 1111 … 1111 1111**	**$\pm$ 1.11..1 x 2^{1-127}**	
…	**…**	**…**	**…**	
0/1	**0000 0001**	**0000 0000 … 0000 0001**	**$\pm$ 1.00..1 x 2^{1-127}**	
0/1	**0000 0001**	**0000 0000 … 0000 0000**	**$\pm$ 1.00..0 x 2^{1-127}**	**~ $\pm$ 1.18 x 10^{-38}**
0/1	0000 0000	1111 1111 … 1111 1111	Denormalized number	
0/1	0000 0000	1111 1111 … 1111 1110		
…	…	…		
0/1	0000 0000	0000 0000 … 0000 0001		
0/1	**0000 0000**	**0000 0000 … 0000 0000**	**0**	**0**

Denormalized Numbers

- Assume significand bit = 3 instead of 23 for simplicity:

0 00000010 001	1.001×2^{-125}
0 00000010 000	1.0×2^{-125}
0 00000001 111	1.111×2^{-126}
0 00000001 110	1.11×2^{-126}
. . .	
0 00000001 000	1.0×2^{-126}
0 00000000 111	1.111×2^{-127}
0 00000000 110	1.11×2^{-127}
. . .	
0 00000000 001	1.001×2^{-127}
0 00000000 000	0



Denormalized Numbers

- Assume significand bit = 3:

0 00000010 001 1.001×2^{-125}

0 00000010 000 1.0×2^{-125}

0 00000001 111 1.111×2^{-126}

0 00000001 110 1.11×2^{-126}

. . .

0 00000001 000 1.0×2^{-126}

0 ***00000000*** 111 1.111×2^{-127} => 0.111×2^{-126}

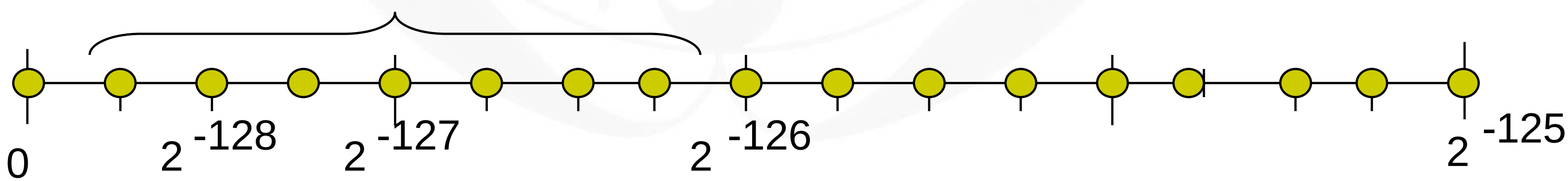
0 ***00000000*** 110 1.11×2^{-127} => 0.110×2^{-126}

. . .

0 ***00000000*** 001 1.001×2^{-127} => 0.001×2^{-126}

Denormalized Numbers

0 00000000 000 0



Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{1-\text{Bias}}$$

  - Smaller than normal numbers
    - allow for gradual underflow, with diminishing precision
  - Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{1-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Range of Numbers Represented by IEEE 754

sign	Exponent	Significand		
0/1	1111 1111	1111 1111 … 1111 1111	NaN	
0/1	1111 1111	…		
0/1	1111 1111	0000 0000 … 0000 0001		
0/1	1111 1111	0000 0000 … 0000 0000	± Infinity	
0/1	**1111 1110**	**1111 1111 … 1111 1111**	**$\pm$ 1.11..1 x $2^{254-127}$**	**~ $\pm$ 3.40 x 10^{38}**
0/1	**1111 1110**	**1111 1111 … 1111 1110**	**$\pm$ 1.11..0 x $2^{254-127}$**	
…	**…**	**…**	**…**	
0/1	**0000 0010**	**0000 0000 … 0000 0000**	**$\pm$ 1.00..0 x 2^{2-127}**	
0/1	**0000 0001**	**1111 1111 … 1111 1111**	**$\pm$ 1.11..1 x 2^{1-127}**	
…	**…**	**…**	**…**	
0/1	**0000 0001**	**0000 0000 … 0000 0001**	**$\pm$ 1.00..1 x 2^{1-127}**	
0/1	**0000 0001**	**0000 0000 … 0000 0000**	**$\pm$ 1.00..0 x 2^{1-127}**	**~ $\pm$ 1.18 x 10^{-38}**
0/1	0000 0000	1111 1111 … 1111 1111	**$\pm$ 0.11..1 x 2^{1-127}**	
0/1	0000 0000	1111 1111 … 1111 1110	**$\pm$ 0.11..0 x 2^{1-127}**	
…	…	…	…	
0/1	0000 0000	0000 0000 … 0000 0001	**$\pm$ 0.00..1 x 2^{1-127}**	**~ $\pm$ 1.4 x 10^{-45}**
0/1	**0000 0000**	**0000 0000 … 0000 0000**	**0**	**0**

Floating-Point Addition

- Consider a **4-digit** decimal example
  - $9.999 \times 10^1 + 1.610 \times 10^{-1}$

1. Align decimal points
   - Shift number with smaller exponent
   - $9.999 \times 10^1 + 0.016 \times 10^1$
2. Add significands
   - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
3. Normalize result & check for over/underflow
   - 1.0015×10^2
4. Round and renormalize if necessary
   - 1.002×10^2

Floating-Point Addition

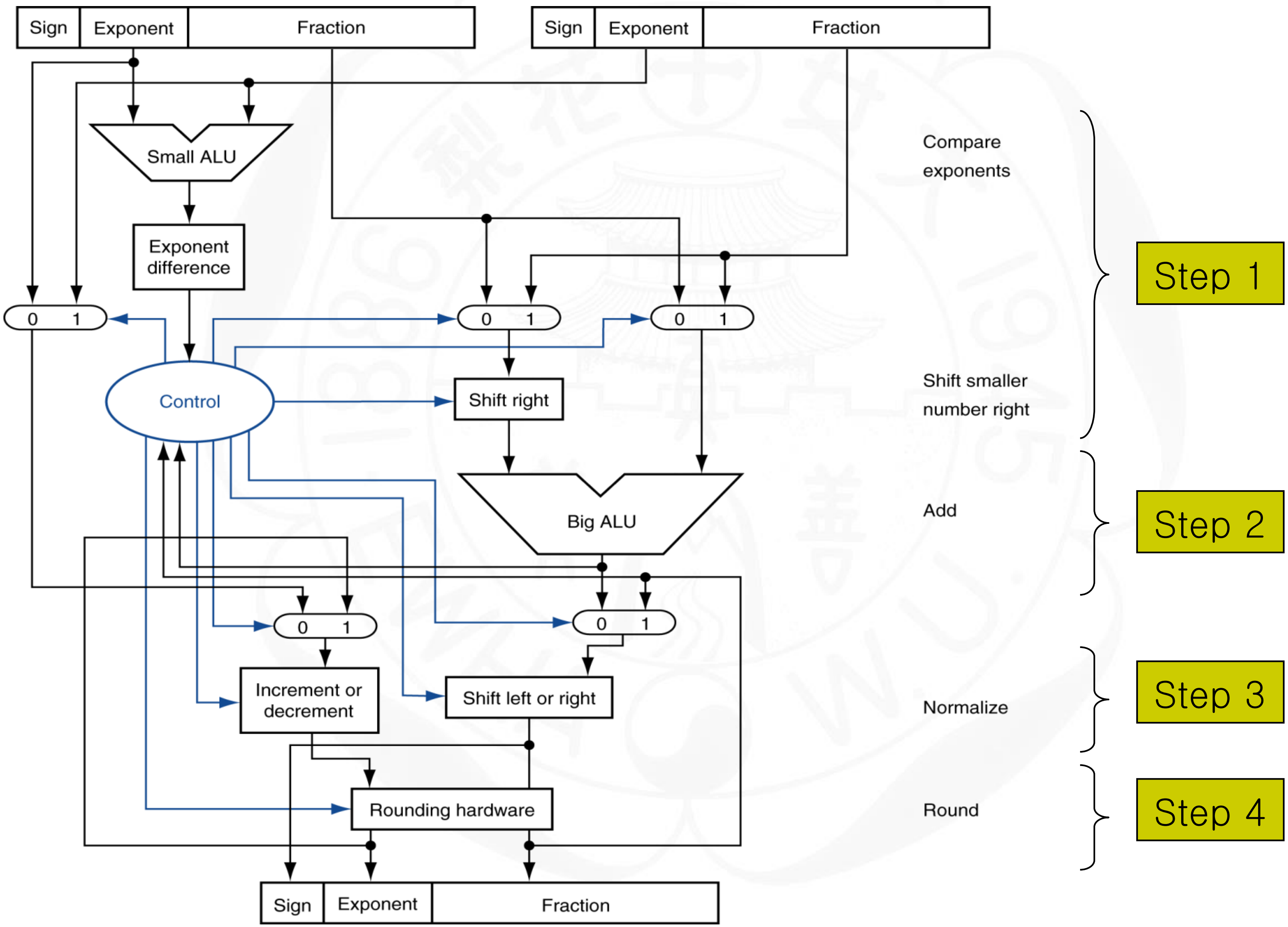
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + (-1.110_2) \times 2^{-2}$ (0.5 + −0.4375)

1. Align binary points
   - Shift number with smaller exponent
   - $1.000_2 \times 2^{-1} + (-0.111_2) \times 2^{-1}$
2. Add significands
   - $1.000_2 \times 2^{-1} + (-0.111_2) \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625$

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Floating-Point Multiplication

- Consider a **4-digit** decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

1. Add exponents
   - New exponent = $10 + (-5) = 5$
2. Multiply significands
   - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
3. Normalize result & check for over/underflow
   - 1.0212×10^{6}
4. Round and renormalize if necessary
   - 1.021×10^{6}
5. Determine sign of result from signs of operands
   - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a **4-digit** binary example
  - $1.000_2 \times 2^{-1} \times (-1.110_2) \times 2^{-2}$ (0.5 × (−0.4375))

1. Add exponents
   - Unbiased: (−1) + (−2) = −3
   - Biased: ((−1) + 127) + ((−2) + 127) = (−3) + 254 − 127 = (−3) + 127
2. Multiply significands
   - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
3. Normalize result & check for over/underflow
   - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
4. Round and renormalize if necessary
   - $1.110_2 \times 2^{-3}$ (no change)
5. Determine sign: +ve × −ve ⇒ −ve
   - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: \$f0, \$f1, … \$f31
  - Paired for double-precision: \$f0/\$f1, \$f2/\$f3, …
    - Release 2 of MIPS ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Example

- C code

```
...
int a, b, c;
...
c = a + b;        add $s0, $s1, $s2;
...
float l, m, n
...
l = m + n;        add.s $f0, $f1, $f2;
```

Example: °F to °C

- C code:

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

  - fahr in $f12, result in $f0, literals(5.0, 9,0, 32.0) in global memory space ($gp, global pointer)
- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)     // $f16 = 5.0
     lwc2  $f18, const9($gp)     // $f18 = 9.0
     div.s $f16, $f16, $f18      // $f16 = 5/9
     lwc1  $f18, const32($gp)    // $f18 = 32
     sub.s $f18, $f12, $f18      // $f18 = fahr-32
     mul.s $f0,  $f16, $f18      // $f0 = $f16 * $f18
     jr    $ra
```

Example for Guard Bit

- Assume 4–bit significand
- Addition: $1.0000 \times 2^{0} + 1.1111 \times 2^{-2}$

$$
\begin{array}{r}
1.0000 \times 2^{0} \\
+\ 0.01111 \times 2^{0} \\
\hline
1.01111
\end{array}
$$

$1.0111\underset{\substack{\uparrow \\ g}}{1}$ rounds to 1.1000 – right! (1.0111, otherwise)

- Multiplication:

$$
\begin{array}{r}
1.1000 \times 2^{0} \\
\times\ 1.0001 \times 2^{-2} \\
\hline
1.10011000 \times 2^{-2}
\end{array}
$$

normalize $1.1001\underset{\substack{\uparrow \\ g}}{1}000 \times 2^{-1}$ rounds to 1.1010×2^{-1}

Rounding Modes

- IEEE Standard has four rounding modes:
  - round to nearest even (default) (반올림)
    - example: base 10
      - 3.56<u>51</u> => 3.57 3.56<u>49</u> => 3.56
      - **3.56<u>50</u> => 3.56 3.57<u>50</u> => 3.58**
    - example: base 2
      - 1.01<u>101</u> => 1.10 1.01<u>011</u> => 1.01
      - **1.01<u>100</u> => 1.10 1.00<u>100</u> => 1.00**
      - 1.00<u>10000001</u> => 1.01
  - round towards plus infinity
  - round towards minus infinity
  - round towards 0

Rounding Modes

- “round to nearest” minimizes the mean error introduced by rounding
- Round Bit is calculated to the right of guard bit
- Sticky Bit is used to determine whether there are any 1 bits truncated below the guard and round bits

Round Bit

- Bit to the right of guard bit needed for accurate rounding
- Example: $1.0000 \times 2^{0} - 1.0001 \times 2^{-2}$
  - guard and round bits shown

$$
\begin{array}{rl}
 & 1.0000 \times 2^{0} \\
- & 0.010001 \times 2^{0} \\
\hline
 & 0.1011\underset{\uparrow g}{1}\underset{\uparrow r}{1} \times 2^{0} \quad \text{Result}
\end{array}
$$

1.01111×2^{-1} Normalize

1.1000×2^{-1} Round

  - Without round bit, result is 1.0111×2^{-1}

Sticky Bit

- 'Round to even' problems
  - need to know if the answer is exactly even or not
- Keep "sticky" bit (S):
  - S = 1 if any bits are off to the right, otherwise S = 0
- Example: $1.0000 \times 2^{0} + 1.0001 \times 2^{-5}$
  - guard, round, and sticky bits shown

$$
\begin{array}{lll}
 & 1.0000 \times 2^{0} & \\
+ & 0.000010 \times 2^{0}\ \boxed{1} & (0.000010\mathbf{001} \times 20 \text{ to be exact}) \\
\hline
 & 1.000010 \times 2^{0}\ \boxed{1} & \text{Result} \\
 & 1.0001 \times 2^{0} & \text{Round to nearest; without S rounds to 1.0000}
\end{array}
$$

Sticky bit

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
  - $(x + y) + z \neq x + (y + z)$

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

x86 FP Architecture

- Originally based on 8087 FP coprocessor
  - 8 × 80-bit extended-precision registers
  - Used as a push-down stack
  - Registers indexed from top of stack: ST(0), ST(1), …
- FP values are 32-bit or 64 in memory
  - Converted on load/store of memory operand
  - Integer operands can also be converted on load/store
- Very difficult to generate and optimize code
  - Result: poor FP performance

Streaming SIMD Extension 2 (SSE2)

- Intel adds 144 new instructions to SSE in 2001
- Adds 4 × 128-bit registers
  - Extended to 8 registers in AMD64/EM64T
- Can be used for multiple FP operands
  - 2 × 64-bit double precision
  - 4 × 32-bit double precision
  - Instructions operate on them simultaneously
    - Single-Instruction Multiple-Data (SIMD)