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

$\Rightarrow +2.000 \times 10^{-7}$

$\Rightarrow +9.8702 \times 10^{11}$

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

CSULB

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)

CSULB

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: 1.0 ≤ |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 = 1023

CSULB

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest absolute value
  - Exponent: 00000001
    ⇒ actual exponent = 1 − 127 = −126 (bias)
  - Fraction: 000…00 ⇒ significand = 1
    1.0×2^{-126}
- Largest absolute value
  - Exponent: 11111110
    ⇒ actual exponent = 11111110 − 127 = 127
  - Fraction: 111…11 ⇒ significand ≈ 1.1111~1 ≈ 2.0
    $\simeq 2.0 \times 2^{127}$

0 1 inf

S	Exponent	Fraction

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

2^E ↑ E → inf

CSULB

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest absolute value
  - Exponent: 0000 ~ 0001
    ⇒ actual exponent = 1 − 1023
  - Fraction: 000…00 ⇒ significand = 1.0
    1.0×2^{-1022}
- Largest absolute value
  - Exponent: 1111 ~ 1110
    ⇒ actual exponent = 2046 − 1023 = 1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
    $\sim 2.0 \times 2^{1023}$

S	Exponent	Fraction

$$x = (-1)^S \times (\underbrace{1+\text{Fraction}}_{1.1111111 \sim 1111}) \times 2^{(\text{Exponent}-\text{Bias})}$$

Denormal Numbers

- Exponent = 000...0 ⇒ hidden bit is 0
  - Denormal with fraction = 000...0

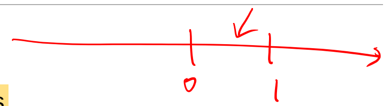
$$x = (-1)^S \times (0+0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Sign	Exponent (e)	Fraction (f)	Value
0	00⋯00	00⋯00	+0
1	00⋯00	00⋯00	−0

Denormal Numbers

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



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

$(-1)^S \times (\cancel{1}^{0} + \text{Fract}) \times 2^E$

  - Smaller than normal numbers
    - allow for gradual underflow, with diminishing precision

Sign	Exponent (e)	Fraction (f)	Value
0	00⋯00	00⋯01 ⋮ 11⋯11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
1	00⋯00	00⋯01 ⋮ 11⋯11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$

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, 0.0 * ∞
  - Can be used in subsequent calculations

Floating-Point Precision

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

$\log_{10} 10^3$

$= 3 \log_{10} 10 = 3$

CSULB

Floating-Point Example

- What number is represented by the single-precision float

$\underbrace{1}_{E}\underbrace{10000001}\underbrace{01000\ldots00}_{F}$

- S = 1
- Fraction = 010 ~ 00
- Exponent = 100 ~ 01 = 129
- $x = (-1)^1 \times (1 + 0.010\sim00) \times 2^{129-127} = -(1.01_2) \times 2^2$

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

$101.01_2 = 1\times2^2 + 0\times2^1 + 1\times2^0 + 0\times2^{-1} + 1\times2^{-2}$

$= 4 + 1 + \frac{1}{4} = 5 + 0.25$

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

CSULB

Floating-Point Example

- Represent –0.75
  - –0.75 = $-0.5-0.25 = -1\times\frac{1}{2} - 1\times\frac{1}{4} = -0.11_2 = -1.1\times 2^{-1}$
  - S = 1
  - Fraction = 1000 ~00
  - Exponent = $(x-127) = -1 = 126 = 01111110$ (single)
    - Single:
    - Double: $0\underbrace{11\sim1}_{9}10$
- Single:
- Double: 0

CSULB

Floating-Point Addition

- Consider a 4-digit decimal example
  - 9.999 × 10^1 + 1.610 × 10^{-1}
- Align decimal points
  - Shift number with smaller exponent

$9.999\times10^1 + 0.0161\times10^1$

- Add significands

10.015×10^1

- Normalize result & check for over/underflow

10.015×10^1

- Round and renormalize if necessary $10.02\times10^1 = 1.002\times10^2$

CSULB

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)
- Align binary points
  - Shift number with smaller exponent

$1.000 \times 2^{-1} + -0.111 \times 2^{-1}$

- Add significands

0.001×2^{-1}

- Normalize result & check for over/underflow

1.000×2^{-4}

- Round and renormalize if necessary

1.000×2^{-4}

CSULB

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

CSULB

Floating-Point Multiplication

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

$10^{10} \times 10^{-5} = 10^{5}$

- Multiply significands

$1.110 \times 9.200 = 10.212$

- Normalize result & check for over/underflow

1.0212×10^{6}

- Round and renormalize if necessary

1.021×10^{6}

- Determine sign of result from signs of operands

1.021×10^{6}

CSULB

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)
- Add exponents
  - Unbiased: $2^{-1} \times 2^{-2} = 2^{-3}$
  - Biased: $2^{124-127}$
- Multiply significands

$1.000 \times 1.110 = 1.110$

- Normalize result & check for over/underflow

1.110×2^{-3}

- Round and renormalize if necessary

1.110×2^{-3}

- Determine sign: +ve × –ve ⇒ –ve

$+ \quad - \quad -$ $\qquad -1.110 \times 2^{-3}$

CSULB

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 $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

CSULB

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)`

CSULB

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`

CSULB

FP 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 in global memory space
- Compiled MIPS code:

```
f2c: lwc1   $f16, const5($gp)
     lwc2   $f18, const9($gp)
     div.s  $f16, $f16, $f18
     lwc1   $f18, const32($gp)
     sub.s  $f18, $f12, $f18
     mul.s  $f0,  $f16, $f18
     jr     $ra
```

CSULB

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(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

CSULB

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - $\frac{4195835}{3145727} = 1.33382044 \neq 1.33373902$
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

CSULB

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

CSULB

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent

CSULB