# CSE3421
# Computer Architecture

# Floating-point Arithmetic

Xiaodong Zhang

Chapter 3.5

# Representing Big (and Small) Numbers

- What if we want to encode the approx. age of the earth?

  4,600,000,000 or $4.6 \times 10^9$ (scientific notation)

  or the weight in kg of one a.m.u. (atomic mass unit)

  0.00000000000000000000000000166 or $1.6 \times 10^{-27}$ (scientific notation)

  There is no way we can encode either of the above in a 32-bit integer.

- Floating point representation $(-1)^{sign} \times F \times 2^E$
  - Still have to fit everything in 32 bits (single precision)

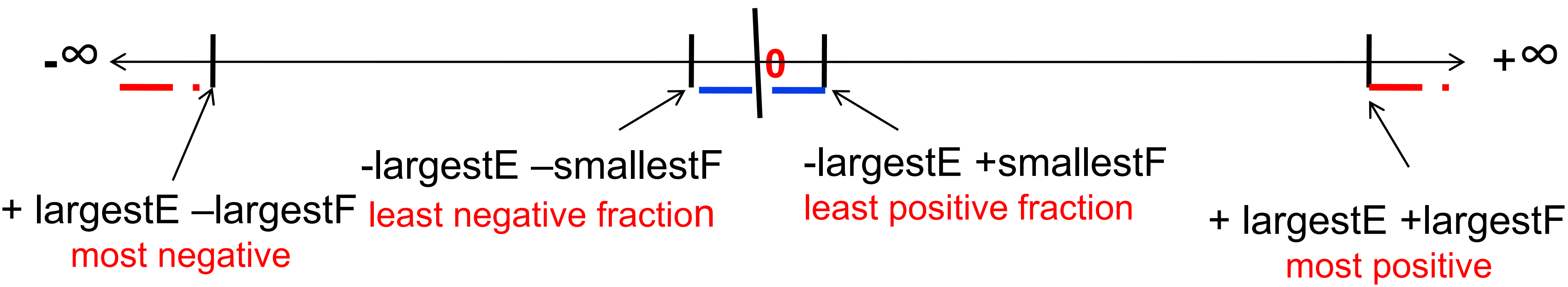
| s | E (exponent) | F (fraction) |
|---|---|---|
| 1 bit | 8 bits | 23 bits |

  - The base (2, *not* 10) is hardwired in the design of the FPALU
  - More bits in the fraction (F) or the exponent (E) is a trade-off between precision (accuracy of the number) and range (size of the number)

# Exception Events in Floating Point

- **Overflow** (floating point) happens when a positive exponent becomes too large to fit in the exponent field

- **Underflow** (floating point) happens when a negative exponent becomes too large to fit in the exponent field




- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - **Double precision** – takes two MIPS words

| s | E (exponent) | F (fraction) |
|---|---|---|
| 1 bit | 11 bits | 20 bits |

| F (fraction continued) |
|---|
| 32 bits |

# Biased notation in binary representation

- Besides 2's complement representation for negative and positive number, another one is called **Biased Notation**
- For a given sequence of bits, the biased notation uses the smallest number to represent the most negative value and the biggest number to resent the most positive value
- For 8-bit Exponent, we have 0-255 possibilities. 0 is used for true zero
  - 00000001 (1) is used to represent most negative number (-126)
  - …
  - 01111110 (126) is the least negative number (-1)
  - 01111111 (127) is **not** used
  - 10000000 (128) is the least positive number  (1)
  - …
  - 11111110 (254) represents the most positive number (127)
  - 11111111 (255) is **not** used

# Biased notation in binary representation (continued)

- "Bias" is the number to be subtracted from the normal value in the bit sequence to determine the real value

- IEEE 754 uses **127** as bias, thus
  - 00000001: 1-127=-126 is the most negative exponent
  - 11111110: 254-127 = 127 is the most positive exponent

- 8 bit exponent represents a number range of 0-255
  - 0 is used for true 0
  - 01111111 (127) is **not** used
  - 11111111 (255) is **not** used
  - a total of **254** values are represented (including 0):
    - 127 positive numbers and 126 negative numbers

- In contrast, 8-bit 2's compliment representation (**256** values)
  - Range: $-2^7$ to $+2^7-1$ = -128 to 127 (128 -numbers, and 127 +numbers)
  - 2 more negative numbers due to no unused bit sequence

# IEEE Standard for Floating-Point Arithmetic (IEEE 754)

- Most computers use IEEE 754: $(-1)^{sign} \times (1+F) \times 2^{E-bias}$
  - Formats for both single and double precision
  - F is stored in normalized format where the most significant bit in F is 1 (so there is no need to store it!) – called the hidden bit
  - E is in biased notation where the bias is -127 for 8-bit of E (-1023 for double precision for 11-bits of E), the most negative is 00000001 = $2^{1-127} = 2^{-126}$ and the most positive is 11111110 = $2^{254-127} = 2^{+127}$
  - Binary bits for -E: E=-126 (1), E=-125(2), … E=-1 (126)
  - Binary bits for +E: E= +127 (254), E=+126(253), … E=+1(128)

- Examples (in normalized format)
  - Smallest+: 0 00000001 1.00000000000000000000000=$1 \times 2^{1-127} = 1 \times 2^{-126}$
  - Zero: 0 00000000 00000000000000000000000 = true 0
  - Largest+: 0 11111110 1.11111111111111111111111 = $(1+1-2^{-23}) \times 2^{254-127} = (1+1-2^{-23}) \times 2^{127}$
  - $1.0_2 \times 2^{-1}$ = 0 01111110 1.00000000000000000000000
  - $0.75_{10} \times 2^{4}$ = 0 10000010 1.10000000000000000000000

# An Explanation of the last example

**EXAMPLE**

Show the IEEE 754 binary representation of the number $-0.75_{\text{ten}}$ in single and double precision.

**ANSWER**

The number $-0.75_{\text{ten}}$ is also

$$-3/4_{\text{ten}} \text{ or } -3/2^2{}_{\text{ten}}$$

It is also represented by the binary fraction

$$-11_{\text{two}}/2^2{}_{\text{ten}} \text{ or } -0.11_{\text{two}}$$

In scientific notation, the value is

$$-0.11_{\text{two}} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{\text{two}} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{\text{two}} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{\text{two}}) \times 2^{(126-127)}$$

The single precision binary representation of $-0.75_{\text{ten}}$ is then

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

1 bit | 8 bits | 23 bits

## How do we represent -3.75?

- 3 is 11 in binary number
- .75 is .11 in binary number
- -3.75 = -(3+0.75) => -11.11 in binary number
- To follow the IEEE 754 format, we shift one integer bit to left
- $3.75 = 1.111 * 2^1$

- Sign = 1, E = 128 (for 1), F= 1110000..00
- 1 11111110 **1**. 111000000…00

## How do we represent 0.1 in floating point format?

- In integer world there is a one-to-one mapping between a decimal number and a binary number
- In floating point world, this one-to-one relationship may not always exist
- An example of existing: $3.75 = 1.111 * 2^{1}$
- How about $0.1 = 1.0 * 10^{-1}$ (10 cents) ?
- 0.1 in binary: 1/1010 = 0.00011001100110011001100110011 …
- In IEEE 754 format: **1**.10011001100110011001101**1** $* 2^{-4}$
- Hidden bit: **1**, rounding bit: **1**
- **10 cents can not be represented exactly in computer!**

## The usage of the unused number

- E= 11111111 (255) is not used
  - If F fraction is 0, and E=11111111, it represents infinity
  - If F fraction is non-zero, it represents an invalid number

# IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for the results of invalid operations such as 0/0
  - True zero is the bit string all zero

| Single Precision | | Double Precision | | Object Represented |
|---|---|---|---|---|
| E (8) | F (23) | E (11) | F (20+32=52) | |
| 0000 0000 | 0 | 0000 … 0000 | 0 | true zero (0) |
| 0000 0000 | nonzero | 0000 … 0000 | nonzero | ± denormalized number (±1.XYZ) |
| +127 to -126 | anything | +1023 to -1022 | anything | ± normalized floating point number |
| 1111 1111 | + 0 (S=0) | 1111 … 1111 | - 0 (S=1) | ± infinity |
| 1111 1111 | nonzero | 1111 … 1111 | nonzero | not a number (NaN) |

# Number rounding inside a computer

- If we do the following addition with only three significant decimal digits:

  $2.56 * 10^0 + 2.34 * 10^2$

  - we must shift the smaller number to the right to align the exponents: $0.0256 * 10^2 + 2.34 * 10^2$
  - Considering three significant digits, the computer addition of the floating part is: 0.02 + 2.34, and the result is 2.36 by missing two digits

- IEEE 754 keeps two extra bits called **guard** and **round** bits
  - With these two bits, we are able to do conduct:

  0.0256 + 2.3400 = 2.3656

- What happens if we still could not hold bits after shifting?

  $A = 5.01 * 10^{-1} + 2.34 * 10^2 = 0.0050 + 2.34$ (missing 1 digit)

  - IEEE 754 adds **sticky** bit, setting to 1 if 1s are shifted off
  - if sticky bit = 0, round to nearest even A = 2.34, otherwise A = 2.35

# Round to "nearest-even" in binary (IEEE 754)

Conditions to round

- If the least significant bit (LSB) is 0, round-up when bits beyond the LSB > 100…00 (>half), otherwise round-down (drop the bits)
- If the least significant bit (LSB) is 1, round-up when bits beyond the LSB >= $100\ldots0_2$ (>=half), otherwise round-down (drop the bits)
- Examples:
  - 10.00011 round to 10.00 (less than ½, 3<4)
  - 10.00110 round to 10.01 (greater than ½, 6>4)
  - 10.11100 round to 11.00 (round up, 4=4)
  - 10.10100 round to 10.10 (round down, 4!>4)

# Support for Accurate Arithmetic

- IEEE 754 FP rounding modes
  - Always round up (toward +∞)
  - Always round down (toward -∞)
  - Truncate
  - Round to nearest even

- Rounding (except for truncation) requires the hardware to include extra F bits during calculations
  - Guard bit – used to provide one F bit to hold possible overflow nominalization
  - Round bit – additional bit to improve rounding accuracy
  - Sticky bit – used to support Round to nearest even decision; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

# Floating Point Addition Example

- Add

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

- Step 0: Hidden bits restored in the representation above
- Step 1: Shift fractions with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)
- Step 2: Add fractions after normalization (different signs)
  $1.0000 \times 2^{-1} + (-0.111) \times 2^{-1} = 1.0000 - 0.111 = 0.001$
- Step 3: Normalize sum (left shift), checking for exponent underflow
  $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = .. = 1.000 \times 2^{-4}$
- Step 4: The sum is already rounded, so we're done
- Step 5: Re-hide the hidden bit before storing

# Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

The hidden bit plus the floating point fraction < 1.999…

- Step 0: Restore the hidden bit in F1 and in F2
- Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming $E1 \geq E2$) keeping track of (three of) the bits shifted out in G R and S
- Step 2: Add the resulting F2 to F1 to form F3
- Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
  - If F1 and F2 have the same sign $\rightarrow$ F3 $\in [1,4)$ $\rightarrow$ 1 bit right shift F3 and increment E3 (check for overflow)
  - If F1 and F2 have different signs $\rightarrow$ F3 may require *many* left shifts each time decrementing E3 (check for underflow)
- Step 4: Round F3 and possibly normalize F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

# Floating Point Multiplication Example

- Multiply

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

  - Step 0: Hidden bits restored in the representation above
  - Step 1: Add the exponents: -1 + (-2) = -3
    Determine the sign: 1
  - Step 2: Multiply the fractions
    1.0000 x 1.110 = 1.110000
  - Step 3: Normalize the product, checking for exp over/underflow
    $1.110000 \text{ x } 2^{-3}$ is already normalized
  - Step 4: The product is already rounded, so we're done
  - Step 5: Rehide the hidden bit before storing

# Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \text{ x } (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

$$(F3 < 1.9999\ldots * 1.9999\ldots) \in [1,4)$$

  - Step 0: Restore the hidden bit in F1 and in F2
  - Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3

    also determine the sign of the product (which depends on the sign of the operands (most significant bits))
  - Step 2: Multiply F1 by F2 to form a double precision F3
  - Step 3: Normalize F3 (so it is in the form 1.XXXXX …)
    - Since F1 and F2 come in normalized $\rightarrow$ F3 $\in [1,4)$ $\rightarrow$ 1 bit right shift F3 and increment E3
    - Check for overflow/underflow
  - Step 4: Round F3 and possibly normalize F3 again
  - Step 5: Rehide the most significant bit of F3 before storing the result

# MIPS Floating Point Instructions

- MIPS has a separate Floating Point Register File (`$f0, $f1, …, $f31`) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them by a coprocessor (cl)

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

- And supports IEEE 754 single

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

and 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`

# 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% |

- The frequency of reading (lw) is about 3 times more than writing (sw)