CSE213

Microcontroller Programming

Floating Point Numbers

Introduction

- In this chapter, representation of real numbers in processors is explained.
- The floating point (FP) representation is introduced.
- Widely preferred floating point number standard, IEEE 754, is described.
- FP arithmetic is briefly explained.

Chapter Objectives

Upon completion of this chapter, you will be able to:

- Explain the reason of why floating point numbers are needed.
- Convert between floating point and real representation of numbers.
- Perform FP arithmetic.

Floating Point

- In high level languages, you can define the type of variables such as:
  - int
  - float
  - double
  - etc.
- However, this is not directly possible in assembly language.
- Binary numbers may have different meaning based on the representation you use.

Floating Point

- The floating-point representation is one way to represent real numbers in binary form.
- Numbers are in general represented approximately to a fixed number of significant digits and scaled using an exponent.
- The base for the scaling is normally 2, 10 or 16.
- The typical number that can be represented exactly is of the form:

  $$\text{significant digits} \times \text{base}^{\text{exponent}}$$

- The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number.

Floating Point

- Though there are different floating point representations, the most commonly encountered representation is IEEE 754 Standard.
- The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values.
- The speed of floating-point operations is an important measure of performance for computers in many application domains. It is measured in FLOPS.

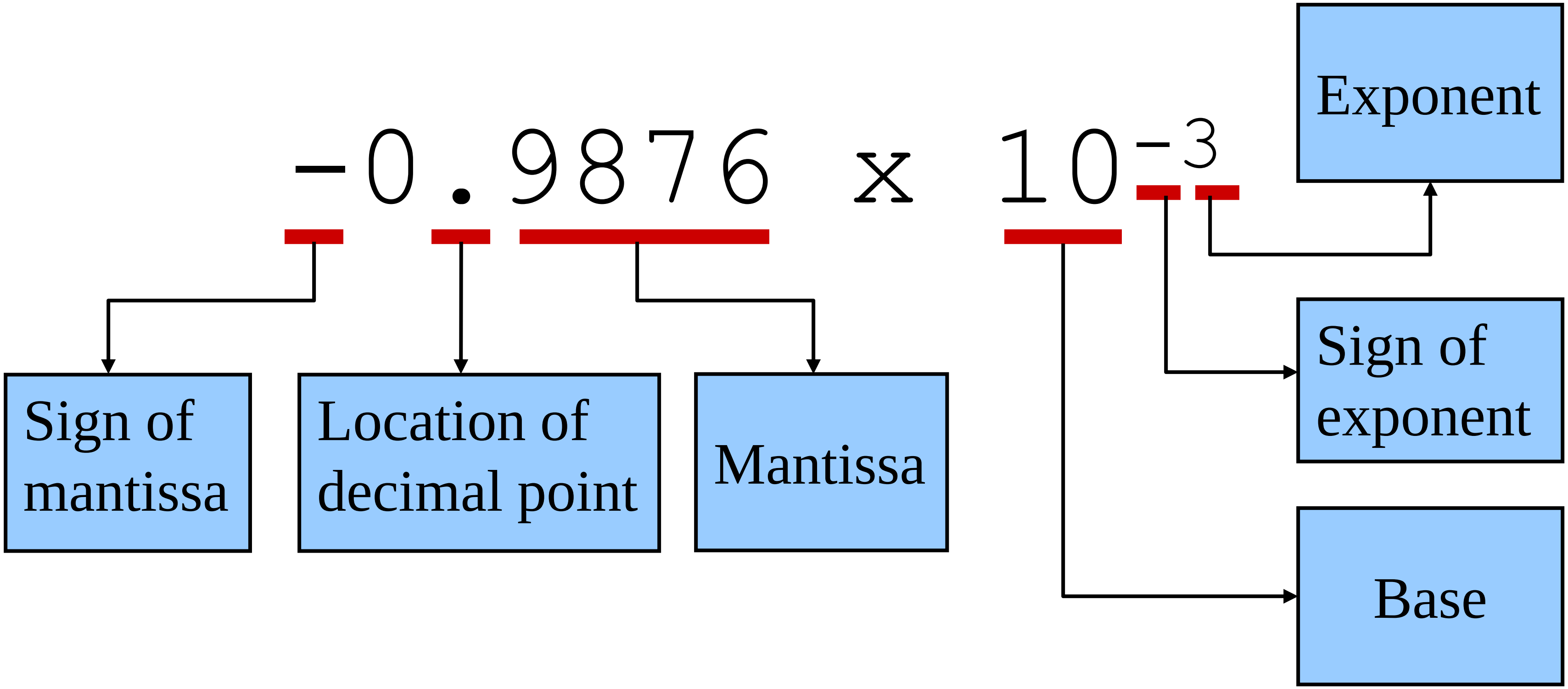
Exponential Notation

- The following are equivalent representations of 1,234

123,400.0	x	10^{-2}
12,340.0	x	10^{-1}
1,234.0	x	10^{0}
123.4	x	10^{1}
12.34	x	10^{2}
1.234	x	10^{3}
0.1234	x	10^{4}

The representations differ in that the decimal place – the “point” -- “floats” to the left or right (with the appropriate adjustment in the exponent).

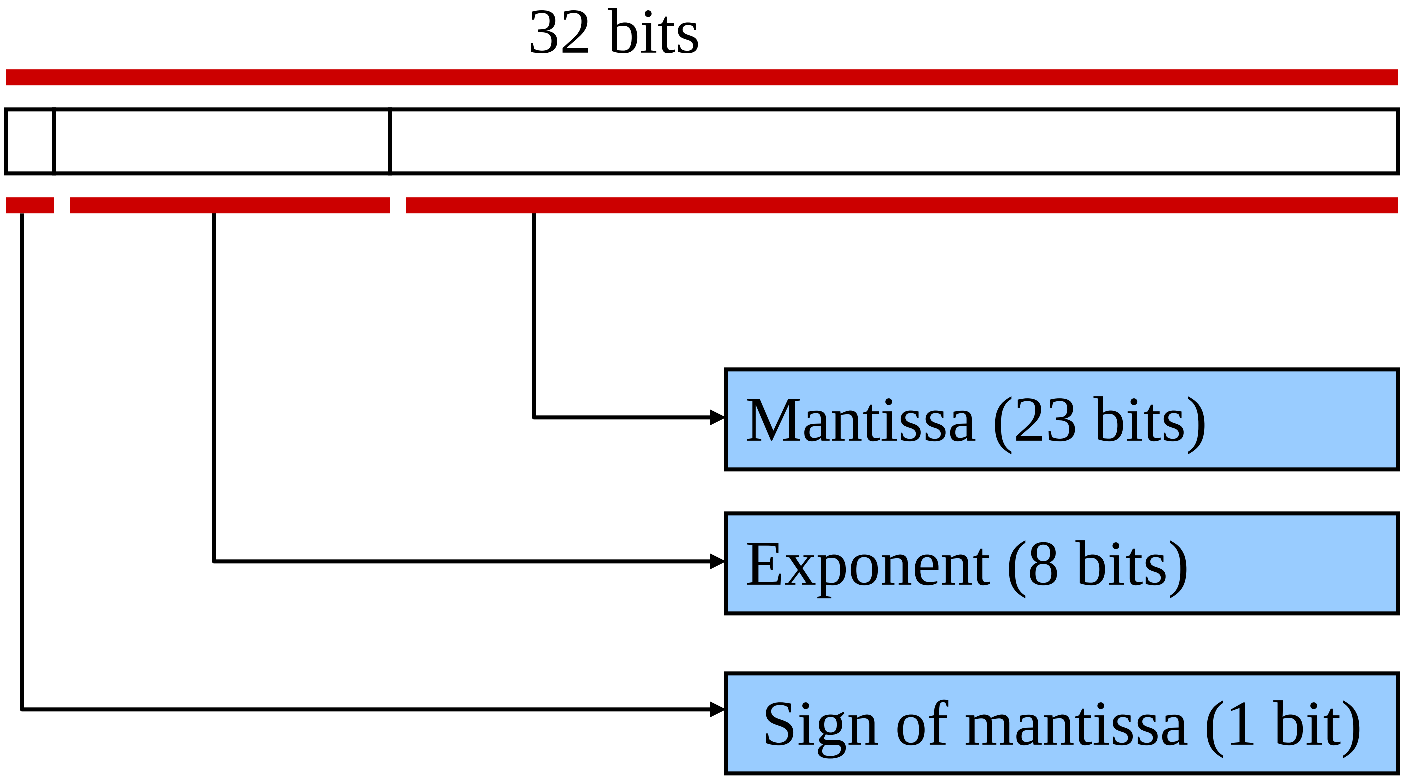
Parts of a Floating Point Number



IEEE 754 Standard

- Most common standard for representing floating point numbers
- Single precision: 32 bits, consisting of:
  - Sign bit (1 bit)
  - Exponent (8 bits)
  - Mantissa (or fraction) (23 bits)
- Double precision: 64 bits, consisting of:
  - Sign bit (1 bit)
  - Exponent (11 bits)
  - Mantissa (or fraction) (52 bits)

Single Precision Format



Normalization

- The mantissa is *normalized*
- Has an implied decimal place on left
- Has an implied "1" on left of the decimal place
- E.g.,
  - Mantissa → `10100000000000000000000`
  - Represents… $1.101_2 = 1.625_{10}$

Excess Notation

- To include exponents, “excess” notation is used
- Single precision: excess 127
- Double precision: excess 1023
- The value of the exponent stored is larger than the actual exponent
- E.g., excess 127,
  – Exponent → `10000111`
  – Represents… `135 - 127 = 8`

Denormalized Values

- Condition
  - `exp` = `000…0`
- Value
  - Exponent value *E* = –*Bias* + 1
  - Significand value *m* = $0.xxx\ldots x_2$
    - **xxx…x**: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - Represents value 0
    - Note that have distinct values +0 and –0
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - “Gradual underflow”

Special Values

- Condition
  - `exp` = `111…1`
- Cases
  - `exp` = `111…1`, `frac` = `000…0`
    - Represents value ∞ (infinity)
    - Operation that overflows
    - Both positive and negative
    - E.g., 1.0/0.0 = –1.0/–0.0 = +∞,   1.0/–0.0 = –∞
  - `exp` = `111…1`, `frac` $\neq$ `000…0`
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - E.g., sqrt(–1), $\infty - \infty$

Example

- Single precision

```
0 10000010 11000000000000000000000
```

1.11_2

$130 - 127 = 3$

$0 =$ positive mantissa

$$+1.11_2 \times 2^3 = 1110.0_2 = 14.0_{10}$$

Hexadecimal

- It is convenient and common to represent the original floating point number in hexadecimal
- The preceding example…

0 100	0001	0 110	0000	0000	0000	0000	0000
4	1	6	0	0	0	0	0

Converting from Floating Point

- E.g., What decimal value is represented by the following 32-bit floating point number?

$$C17B0000_{16}$$

- Step 1
  - Express in binary and find S, E, and M

$\mathtt{C17B0000}_{16} =$

$\underbrace{\mathtt{1}}_{S}\ \underbrace{\mathtt{10000010}}_{E}\ \underbrace{\mathtt{11110110000000000000000}}_{M}{}_{2}$

1 = negative
0 = positive

$C17B0000_{16} =$

$1 \; 10000010 \; 11110110000000000000000_2$

S E M

- Step 2
  - Find “real” exponent, *n*
  - $n = E – 127$

    $= 10000010_2 – 127$

    $= 130 – 127$

    $= 3$

$C17B0000_{16} =$

$$\underbrace{1}_{S}\ \underbrace{10000010}_{E}\ \underbrace{11110110000000000000000}_{M}{}_2$$

- Step 3
  - Put S, M, and *n* together to form binary result
  - (Don't forget the implied "1." on the left of the mantissa.)

$$-1.1111011_2 \text{ x } 2^n =$$

$$-1.1111011_2 \text{ x } 2^3 =$$

$$-1111.1011_2$$

$C17B0000_{16}$ =

$1\ 10000010\ 11110110000000000000000_2$

S E M

- Step 4
  - Express result in decimal

-1111.1011_2

-15

$2^{-1} = 0.5$

$2^{-3} = 0.125$

$2^{-4} = 0.0625$

0.6875

Answer: -15.6875

Converting <u>to</u> Floating Point

- E.g., Express 36.5625_{10} as a 32-bit floating point number (in hexadecimal)

- Step 1
  - Express original value in binary

$36.5625_{10} =$

100100.1001_2

- Step 2
  - Normalize

$$100100.1001_2 =$$

$$1.001001001_2 \times 2^5$$

- Step 3
  - Determine S, E, and M

$$\underbrace{+}_{S}1.\underbrace{001001001}_{M}{}_2 \times 2^{\underbrace{5}_{n}}$$

$E = n + 127$
$= 5 + 127$
$= 132$
$= 10000100_2$

S = 0 (because the value is positive)

$$\underbrace{+}_{S}1.\underbrace{001001001}_{M}{}_2 \times 2^{\underbrace{5}_{n}}$$

- Step 4
  - Put S, E, and M together to form 32-bit binary result

$$\underbrace{0}_{S}\ \underbrace{10000100}_{E}\ \underbrace{00100100100000000000000}_{M}{}_2$$

$$\underbrace{+}_{S}1.\underbrace{001001001}_{M}{}_2 \times 2^{\underbrace{5}_{n}}$$

- Step 5
  - Express in hexadecimal

$0\ 10000100\ 00100100100000000000000_2 =$

$0100\ 0010\ 0001\ 0010\ 0100\ 0000\ 0000\ 0000_2 =$

$4\quad 2\quad 1\quad 2\quad 4\quad 0\quad 0\quad 0_{16}$

Answer: 42124000_{16}

Floating Point Operations

- Conceptual View
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into `frac`
- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
– Zero	**$1.00**	**$1.00**	**$1.00**	**$2.00**	**–$1.00**
– Round down (-∞)	**$1.00**	**$1.00**	**$1.00**	**$2.00**	**–$2.00**
– Round up (+∞)	**$2.00**	**$2.00**	**$2.00**	**$3.00**	**–$1.00**
– Nearest Even (default)	**$1.00**	**$2.00**	**$2.00**	**$2.00**	**–$2.00**

Note:

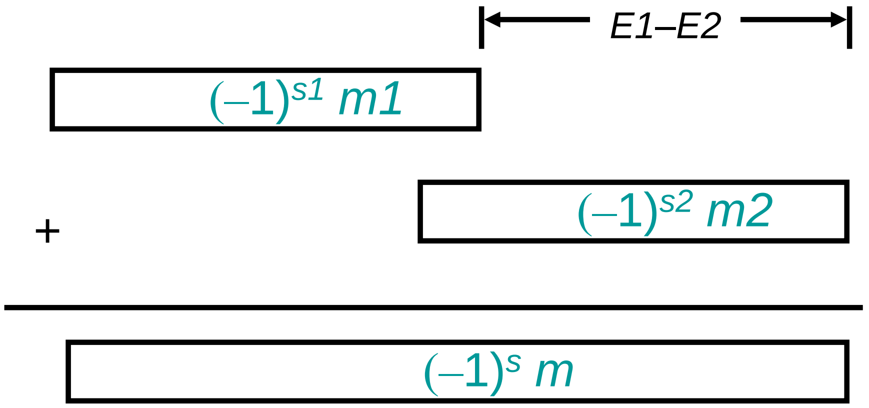
1. **Round down: rounded result is close to but no greater than true result.**
2. **Round up: rounded result is close to but no less than true result.**

FP Multiplication

- Operands
  $(-1)^{s1}\ M1\ \ 2^{E1}$
  $(-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result
  $(-1)^{s}\ M\ \ 2^{E}$
  - Sign ***s***: ***s1 ^ s2***
  - Significand ***M***: ***M1 * M2***
  - Exponent ***E***: ***E1 + E2***
- Fixing
  - If $M \geq 2$, shift ***M*** right, increment ***E***
  - If ***E*** out of range, overflow
  - Round ***M*** to fit `frac` precision
- Implementation
  - Biggest chore is multiplying significands

FP Addition

- Operands
  - $(-1)^{s1}\ M1\ \ 2^{E1}$
  - $(-1)^{s2}\ M2\ \ 2^{E2}$
  - – Assume ***E1*** > ***E2***



- Exact Result
  - $(-1)^{s}\ M\ \ 2^{E}$
  - – Sign ***s***, significand ***M***:
    - Result of signed align & add
  - – Exponent ***E***: ***E1***
- Fixing
  - – If $M \geq 2$, shift ***M*** right, increment ***E***
  - – if $M < 1$, shift ***M*** left ***k*** positions, decrement ***E*** by ***k***
  - – Overflow if ***E*** out of range
  - – Round ***M*** to fit `frac` precision

FP Operations on Intel Microprocessor

- Intel 80X87 family coprocessors support floating point numbers.
- 80X87 is an external integrated circuit designed to operate concurrently with the microprocessor.
  - 80486DX–Core2 have internal versions of 80387.
- The processor executes all normal instructions and 80X87 executes coprocessor instructions.
  - 80X87 executes 68 different instructions

SUMMARY

- Representation of real numbers in processors is explained.
- FP representation is introduced.
- IEEE 754 standard is described.
- FP arithmetic is briefly explained.

Extra References

- Lecture Notes, “Introduction to Information Technologies”, York University
- Lecture Notes, Course CS 213.

The Intel Microprocessors: 8086/8088, 80186/80188, 80286, 80386, 80486 Pentium, Pentium Pro Processor, Pentium II, Pentium, 4, and Core2 with 64-bit Extensions Architecture, Programming, and Interfacing, Eighth Edition
Barry B. Brey