Floating Point

CSCI3240: Lecture 4 and 5

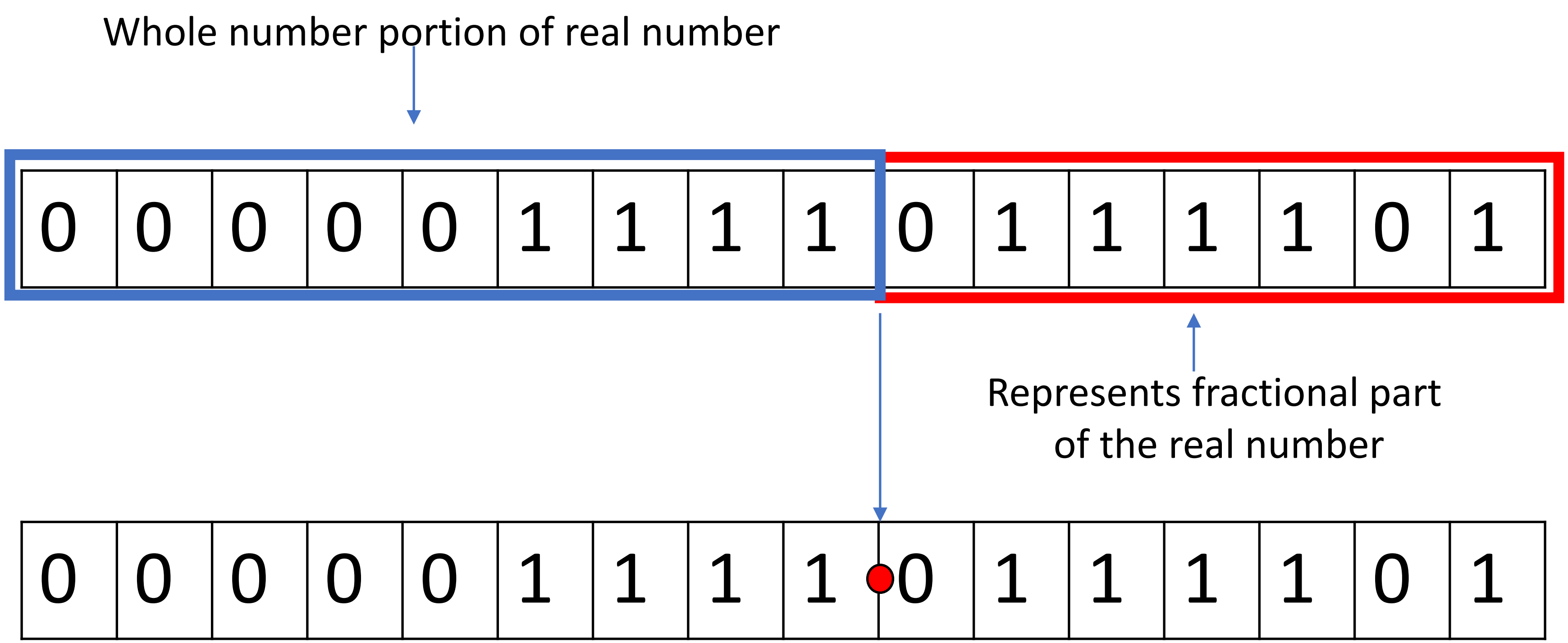
Dr. Arpan Man Sainju

Middle Tennessee State University

Representing Real Numbers in Binary

- What is 1111.0111101_2?
- Let's first understand what fixed point binary is.
- Next, we will learn what floating point binary is.

Fixed Point Binary Fractions



- We can use an imaginary binary point to separate whole number from fractional part.
- The position of the binary point is fixed and can’t be moved.
- In this example, 9-bits are used to represent whole number portion and 7 bits are used to represent fractional part.

Converting Fixed Point Binary to Denary

0	0	0	0	0	1	1	1	1.	0	1	1	1	1	0	1
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

Converting the whole portion:

$$= 2^3 + 2^2 + 2^1 + 2^0$$
$$= 8 + 4 + 2 + 1$$
$$= 15$$

Converting the fractional portion:

$$= 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7}$$
$$= \frac{1}{4} + \frac{1}{8} + \frac{1}{32} + \frac{1}{128}$$

$$= 0.414063$$

$$(00001111.0111101)_2 = \ 15 + 0.414063 = \ 15.414063$$

Another Example

1	1	0	0	0	1	1	0	0 •	1	1	0	0	0	0	0
-2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

Converting the whole portion:

$$= -2^8 + 2^7 + 2^3 + 2^2$$
$$= -256 + 128 + 8 + 4$$
$$= -116$$

Converting the fractional portion:

$$= 2^{-1} + 2^{-2}$$
$$= \frac{1}{2} + \frac{1}{4}$$
$$= 0.75$$

$$(110001100.1100000)_2 = \ -116 + .75 = \ -115.25$$

Practice Questions

- The following binary numbers are stored using two's complement in a 12-bit register with 4 bits after the binary point. Convert them into decimal fraction.

- 010001111010

- 111111111111

- 100001110010

Practice Questions

- Using two's complement, convert the following decimal numbers into fixed point binary to be stored in a 12 bit register with 4-bit after the binary point.
- 27.5

0	0	0	1	1	0	1	1	1	0	0	0
-128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625

- -55.75

1	1	0	0	1	0	0	0	0	1	0	0
-128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625

- -1.75

1	1	1	1	1	1	1	0	0	1	0	0
-128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625

Observation

- Given a 4-bit register, with 1-bit before and 3-bit after the binary point, using two's complement, calculate:
- The largest positive number that can be represented is

0	1	1	1
-1	0.5	0.25	0.125

$= 0.5 + 0.25 + 0.125 = 0.875$

- The smallest positive number that can be represented (not including 0)

0	0	0	1
-1	0.5	0.25	0.125

$= 0.125$

Observation

- Given a 4-bit register, with 1-bit before and 3-bit after the binary point, using two's complement, calculate:
- The smallest magnitude negative number that can be represented (closet to 0)

1 •	1	1	1
-1	0.5	0.25	0.125

$= -1 + 0.5 + 0.25 + 0.125 = -0.125$

- The largest magnitude negative number that can be represented

1 •	0	0	0
-1	0.5	0.25	0.125

$= -1$

Summary

- Fixed point binary is used in digital signal processing.
- It is employed when performance is more important than accuracy.
  - Gaming
- Simple and cheaper processor hardware
- Faster processing
- Tradeoff between range and precision
- Some numbers can never be represented accurately. Such as 1/10.

Floating Point Binary

- Standard Scientific Notation
  - $2.99\ x\ 10^{8}$ Speed of light
  - $6.02\ x\ 10^{23}$ Avogadro’s number
  - $1.60\ x\ 10^{-19}$ Charge of electron

Floating Point

- Number written in scientific notation have three components:

$$-1.25 \text{ X } 10^{-5}$$

sign Mantissa Exponent

$$= -0.0000125$$

$$6.0222140857 \text{ X } 10^{23}$$

Sign (by default, positive) Mantissa Exponent

$$= 602221408570000000000000$$

- The number of digits that you are allowed to use in Mantissa governs the precision of values.
- The number of digits available for exponent governs the range. For 2 digits you can only float the point up to 99 places.

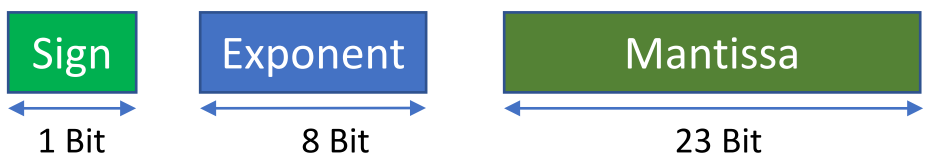
IEEE 754 Standard for Floating-Point Representation

- Number written in scientific notation have three components:

$$-1.25 \text{ X } 10^{-5}$$

sign Mantissa Exponent

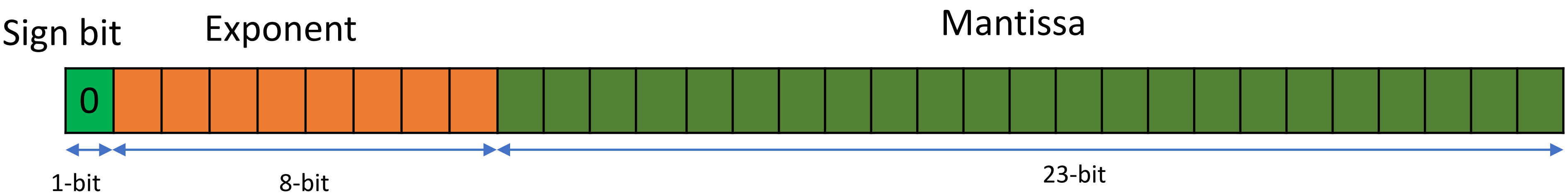
- Three fields:
  - Single precision: 1-bit (sign), 8-bit (exponent), 23-bit (Mantissa) : **32-bits total**



  - Double precision: 1-bit (sign), 11-bit (exponent), 52-bit (Mantissa) : **64-bits total**

Mantissa: fraction part of a floating-point number

IEEE 754 Standard for Single Precision 32-bit floating point binary



Convert the real number 27.236875 into IEEE 7544 standard 32-bit floating point binary

Step 1: Determine the sign bit (0 if positive, 1 if negative)

Sign bit = 0

Step 2: Convert to pure binary

A. Converting whole number part.

$$27 = (11011)_2$$

$$27.2185 = (11011.00111)_2$$

B. Converting the fractional part

0.236875×2	$= 0.47375$	0
0.47375×2	$= 0.875$	0
0.875×2	$= 1.75$	1
0.75×2	$= 1.50$	1
0.50×2	$= 1.00$	1

Stop when product is 1

MIDDLE TENNESSEE STATE UNIVERSITY

I AM true BLUE

IEEE 754 Standard for Single Precision 32-bit floating point binary

Sign bit	Exponent								Mantissa																						
0	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1-bit	8-bit								23-bit																						

Convert the real number 27.2185 into IEEE 754 standard 32-bit floating point binary

Step 3: Normalize to determine the Mantissa and the unbiased exponent

- place the binary point after leftmost 1

$$11011.00111 = 1.\boxed{101100111}\ x\ 2^4 \qquad \textbf{\textit{Unbiased Exponent}} = 4 = 000000100$$

Step 4: Determine the biased exponent (K = number of bits in exponent)

- add bias ($2^{k-1} - 1 = 2^7 - 1$) 127 then convert to an 8-bit unsigned binary integer

$$= 4 + 127 = 131 = (10000011)_2$$

Step 5: Remove the leading 1 from the Mantissa

- remove the leftmost 1

$$\textbf{\textit{Mantissa}} = \boxed{101100111}$$

Note:

- *the left most bit of Mantissa is always going to be 1. (see step 3)*
- *No need to store it.*
- *We get extra 1-bit precision*

Why do we need to add bias to the exponent?

Assume 4-bit exponent				Only +ve exponents	Two's complement	Changing range		IEEE 754 Standard
0	0	0	0	0	0	-5	-9	-7
0	0	0	1	1	1	-4	-8	-6
0	0	1	0	2	2	-3	-7	-5
0	0	1	1	3	3	-2	-6	-4
0	1	0	0	4	4	-1	-5	-3
0	1	0	1	5	5	0	-4	-2
0	1	1	0	6	6	1	-3	-1
0	1	1	1	7	7	2	-2	0
1	0	0	0	8	-8	3	-1	1
1	0	0	1	9	-7	4	0	2
1	0	1	0	10	-6	5	1	3
1	0	1	1	11	-5	6	2	4
1	1	0	0	12	-4	7	3	5
1	1	0	1	13	-3	8	4	6
1	1	1	0	14	-2	9	5	7
1	1	1	1	15	-1	10	6	8

- Slightly favors positive numbers
- Here, bias is 7 because we have 7 -ve numbers.
- We have to add bias to convert the number into unsinged binary for IEEE 754 standard.

- -7 +7 = 0
- -6 +7 = 1
- -5+7 = 2
- 6 + 7 = 13

Exponent bias

IEEE 754 Format	Sign	Exponent	Mantissa	Exponent Bias
32 bit single precision	1 bit	8 bit	23 bits (+1 not stored)	$2^{8-1} - 1 = 127$
64 bit double precision	1 bit	11 bit	52 bits (+1 not stored)	$2^{11-1} - 1023$

Why not use Two's complement instead of IEEE 754 Standard for exponent?

				Two's complement	IEEE 754 Standard
0	0	0	0	0	-7
0	0	0	1	1	-6
0	0	1	0	2	-5
0	0	1	1	3	-4
0	1	0	0	4	-3
0	1	0	1	5	-2
0	1	1	0	6	-1
0	1	1	1	7	0
1	0	0	0	-8	1
1	0	0	1	-7	2
1	0	1	0	-6	3
1	0	1	1	-5	4
1	1	0	0	-4	5
1	1	0	1	-3	6
1	1	1	0	-2	7
1	1	1	1	-1	8

With IEEE 754 standard the computer can easily identify if one number is bigger then another by just looking at the bit pattern.
It is not possible with 2's complement.
See 7 and -8.

Practice Questions

- Convert 0.6875 into IEEE 754 single precision floating point binary
  - Step 1: Determine the sign
  - Step 2: Convert to pure binary
  - Step 3: Normalize for Mantissa and unbiased exponent
  - Step 4: Determine biased exponent
  - Step 5: Remove leading 1 from Mantissa

Practice Question

- Convert -123.84375 into IEEE 754 single precision floating point binary
  - Step 1: Determine the sign
  - Step 2: Convert to pure binary
  - Step 3: Normalize for Mantissa and unbiased exponent
  - Step4: Determine biased exponent
  - Step5: Remove leading 1 from Mantissa

Converting back to decimal

1. Determine the sign in decimal
2. Determine the exponent in decimal
3. Remove the exponent bias
   - Subtract $(2^{k-1} - 1)$, where k is number of bits in exponent field.
4. Convert the Mantissa to decimal
5. Add 1 to the Mantissa and include the sign
6. Compute the final result.

Practice Question

- Convert 01000001001101010000000000000000 into decimal

Practice Question

- Convert 10000001001000110010000000000000 into decimal

Reserved Exponent Values

Exponent Values	Mantissa	Represents
11111111	All zeros	$Infinity$
11111111	Not all zeros	$Not\ a\ number\ (NAN)$
00000000	All zeros	Zero

Floating Point Operations: Basic Idea

- `x +f y = Round(x + y)`
- `x ×f y = Round(x × y)`
- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into Mantissa **`fraction`**

Rounding

- **Rounding Modes (illustrate with $ rounding)**

	\$1.40	**\$1.60**	**\$1.50**	**\$2.50**	**–\$1.50**
Towards zero	\$1	\$1	\$1	\$2	–\$1
Round down ($-\infty$)	\$1	\$1	\$1	\$2	–\$2
Round up ($+\infty$)	\$2	\$2	\$2	\$3	–\$1
Nearest Even (default)	\$1	\$2	\$2	\$2	–\$2

Rounding Binary Numbers

- **Binary Fractional Numbers**
  - "Even" when least significant bit is `0`
  - "Half way" when bits to right of rounding position = `100`…$_2$

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

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

FP Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1$ ^ $s2$
  - Mantissa M: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

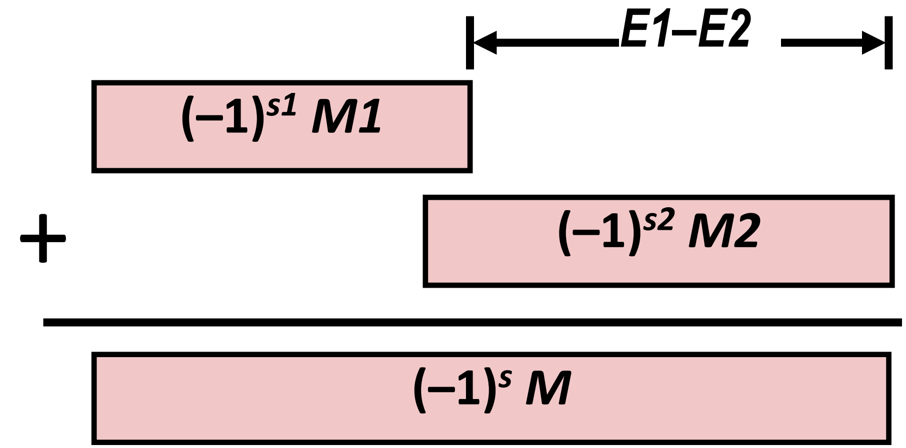
- **Fixing**
  - Normalize (move decimal point after first 1)
  - If *E* out of range, overflow
  - Round *M* to fit Mantissa `fraction` precision

- **Implementation**
  - Biggest chore is multiplying Mantissas

Floating Point Addition

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

Get binary points lined up



- **Exact Result:** $(-1)^{s}\, M\ \ 2^{E}$
  - Sign s, Mantissa M:
    - Result of signed align & add
  - Exponent E: $E1$

- **Fixing**
  - Normalize (move decimal point after first 1)
  - Overflow if E out of range
  - Round M to fit Mantissa `fraction` precision

Mathematical Properties of FP Add

- **Compare to those of Abelian Group**
  - Closed under addition? ***Yes***
    - But may generate infinity or NaN
  - Commutative? ***Yes***
  - Associative? ***No***
    - Overflow and inexactness of rounding
    - `(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`
  - 0 is additive identity? ***Yes***
  - Every element has additive inverse? ***Almost***
    - Yes, except for infinities & NaNs
- **Monotonicity**
  - $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***
    - Except for infinities & NaNs

Mathematical Properties of FP Mult

- **Compare to Commutative Ring**
  - Closed under multiplication? ***Yes***
    - But may generate infinity or NaN
  - Multiplication Commutative? ***Yes***
  - Multiplication is Associative? ***No***
    - Possibility of overflow, inexactness of rounding
    - Ex: `(1e20*1e20)*1e-20= inf`, `1e20*(1e20*1e-20)= 1e20`
  - 1 is multiplicative identity? ***Yes***
  - Multiplication distributes over addition? ***No***
    - Possibility of overflow, inexactness of rounding
    - `1e20*(1e20-1e20)= 0.0`, `1e20*1e20 – 1e20*1e20 = NaN`
- **Monotonicity**
  - $a \geq b$ & $c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
    - Except for infinities & NaNs

Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/Casting**
  - 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
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode

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`