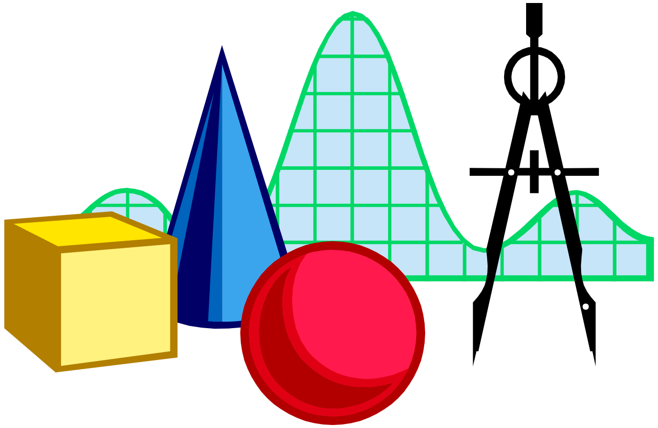
Lecture 3 Floating Point Representations

Floating-point arithmetic



- ☐ We often incur floating-point programming.
  - Floating point greatly simplifies working with large (e.g., 2^{70}) and small (e.g., 2^{-17}) numbers
- ☐ We'll focus on the IEEE 754 standard for floating-point arithmetic.
  - How FP numbers are represented
  - Limitations of FP numbers
  - FP addition and multiplication

Floating-point representation

- IEEE numbers are stored using a kind of scientific notation.

$$\pm \text{mantissa} * 2^{\text{exponent}}$$

- We can represent floating-point numbers with three binary fields: a sign bit s, an exponent field e, and a fraction field f.

- The IEEE 754 standard defines several different precisions.
  - **Single precision** numbers include an 8-bit exponent field and a 23-bit fraction, for a total of 32 bits.
  - **Double precision** numbers have an 11-bit exponent field and a 52-bit fraction, for a total of 64 bits.

Sign

- The sign bit is 0 for positive numbers and 1 for negative numbers.
- But unlike integers, IEEE values are stored in signed magnitude format.

Mantissa

- There are many ways to write a number in scientific notation, but there is always a *unique* normalized representation, with exactly one non-zero digit to the left of the point.

$$0.232 \times 10^{3} = 23.2 \times 10^{1} = 2.32 * 10^{2} = \ldots$$

$$01001 = 1.001 \times 2^{3} = \ldots$$

- What's the normalized representation of 00101101.101 ?

$$00101101.101$$
$$= 1.01101101 \times 2^{5}$$

- What's the normalized representation of 0.0001101001110 ?

$$0.0001101001110$$
$$= 1.110100111 \times 2^{-4}$$

Mantissa

s	e	f

- There are many ways to write a number in scientific notation, but there is always a *unique* normalized representation, with exactly one non-zero digit to the left of the point.

$$0.232 \times 10^3 = 23.2 \times 10^1 = 2.32 * 10^2 = \ldots$$

$$01001 = 1.001 \times 2^3 = \ldots$$

- The field f contains a binary fraction.
- The actual mantissa of the floating-point value is (1 + f).
  - In other words, there is an implicit 1 to the left of the binary point.
  - For example, if f is 01101…, the mantissa would be 1.01101…
- A side effect is that we get a little more precision: there are 24 bits in the mantissa, but we only need to store 23 of them.
- But, what about value 0?

Exponent

s e f

- There are special cases that require encodings
  - Infinities (overflow)
  - NAN (divide by zero)
- For example:
  - Single-precision: 8 bits in e → 256 codes; 11111111 reserved for special cases → 255 codes; one code (00000000) for zero → 254 codes; need both positive and negative exponents → half positives (127), and half negatives (127)
  - Double-precision: 11 bits in e → 2048 codes; 111…1 reserved for special cases → 2047 codes; one code for zero → 2046 codes; need both positive and negative exponents → half positives (1023), and half negatives (1023)

Exponent

s	e	f

- The e field represents the exponent as a biased number.
  - It contains the actual exponent *plus* 127 for single precision, or the actual exponent *plus* 1023 in double precision.
  - This converts all single-precision exponents from -126 to +127 into unsigned numbers from 1 to 254, and all double-precision exponents from -1022 to +1023 into unsigned numbers from 1 to 2046.
- Two examples with single-precision numbers are shown below.
  - If the exponent is 4, the e field will be 4 + 127 = 131 (10000011_2).
  - If e contains 01011101 (93_{10}), the actual exponent is 93 - 127 = -34.
- Storing a biased exponent means we can compare IEEE values as if they were signed integers.

Mapping Between e and Actual Exponent

e		Actual Exponent
0000 0000		Reserved
0000 0001	1-127 = -126	-126_{10}
0000 0010	2-127 = -125	-125_{10}
…		…
0111 1111		0_{10}
…		…
1111 1110	254-127=127	127_{10}
1111 1111		Reserved

Converting an IEEE 754 number to decimal

- The decimal value of an IEEE number is given by the formula:

$$(1 - 2s) * (1 + f) * 2^{e-bias}$$

- Here, the s, f and e fields are assumed to be in decimal.
  - (1 - 2s) is 1 or -1, depending on whether the sign bit is 0 or 1.
  - We add an implicit 1 to the fraction field f, as mentioned earlier.
  - Again, the bias is either 127 or 1023, for single or double precision.

Example IEEE-decimal conversion

- Let’s find the decimal value of the following IEEE number.

1 01111100 11000000000000000000000

- First convert each individual field to decimal.
  - The sign bit s is 1.
  - The e field contains 01111100 = 124_{10}.
  - The mantissa is 0.11000… = 0.75_{10}.
- Then just plug these decimal values of s, e and f into our formula.

$$(1 - 2s) * (1 + f) * 2^{e-bias}$$

- This gives us $(1 - 2) * (1 + 0.75) * 2^{124-127} = (-1.75 * 2^{-3}) = -0.21875$.

Converting a decimal number to IEEE 754

- **What is the single-precision representation of 347.625?**

1. **First convert the number to binary: 347.625 = 101011011.101_2.**
2. **Normalize the number by shifting the binary point until there is a single 1 to the left:**

$101011011.101 \times 2^0 = 1.01011011101 \times 2^8$

3. **The bits to the right of the binary point comprise the fractional field f.**
4. **The number of times you shifted gives the exponent. The field e should contain: exponent + 127.**
5. **Sign bit: 0 if positive, 1 if negative.**

Exercise

- What is the single-precision representation of 639.6875

$$639.6875 = 1001111111.1011_2 = 1.0011111111011 \times 2^9$$

s = 0

e = 9 + 127 = 136 = 10001000

f = 0011111111011

The single-precision representation is:

0 10001000 00111111110110000000000

Examples: Compare FP numbers (<, > ?)

1. 0 0111 1111 110…0 0 1000 0000 110…0

   $+1.11_2 \times 2^{(127-127)} = 1.75_{10}$ $+1.11_2 \times 2^{(128-127)} = 11.1_2 = 3.5_{10}$

0 0111 1111 110…0 0 1000 0000 110…0

+ 0111 1111 < + 1000 0000

directly comparing exponents as unsigned values gives result

2. 1 0111 1111 110…0 1 1000 0000 110…0

   $-f \times 2^{(0111\ 1111)}$ $-f \times 2^{(1000\ 0000)}$

For exponents: 0111 1111 < 1000 0000

So $-f \times 2^{(0111\ 1111)} > -f \times 2^{(1000\ 0000)}$

Special Values (single-precision)

E	F	meaning	Notes
00000000	0…0	0	+0.0 and -0.0
00000000	X…X	Valid number	Unnormalized $=(-1)^{S} \times 2^{-126} \times (0.F)$
11111111	0…0	Infinity	
11111111	X…X	Not a Number	

E	Real Exponent	F	Value
0000 0000	Reserved	000…0	0_{10}
		xxx…x	Unnormalized $(-1)^S$ x 2^{-126} x (0.F)
0000 0001	-126_{10}		Normalized $(-1)^S$ x 2^{e-127} x (1.F)
0000 0010	-125_{10}		
…	…		
0111 1111	0_{10}		
…	…		
1111 1110	127_{10}		
1111 1111	Reserved	000…0	Infinity
		xxx…x	NaN

Range of numbers

- ❑ Normalized (positive range; negative is symmetric)

smallest	00000000100000000000000000000000	$+2^{-126}(1+0) = 2^{-126}$
largest	01111111011111111111111111111111	$+2^{127}(2-2^{-23})$

- ❑ Unnormalized

smallest	00000000000000000000000000000001	$+2^{-126}(2^{-23}) = 2^{-149}$
largest	00000000011111111111111111111111	$+2^{-126}(1-2^{-23})$

In comparison

- The smallest and largest possible 32-bit integers in two’s complement are only -2^{31} and $2^{31} - 1$
- How can we represent so many more values in the IEEE 754 format, even though we use the same number of bits as regular integers?

what’s the next representable FP number?

2^{-126}

0

$+2^{-126}(1+2^{-23})$ **differ from the smallest number by 2^{-149}**

Finiteness

- There *aren't* more IEEE numbers.
- With 32 bits, there are 2^{32}, or about 4 billion, different bit patterns.
  - These can represent 4 billion integers *or* 4 billion reals.
  - But there are an infinite number of reals, and the IEEE format can only represent *some* of the ones from about -2^{128} to $+2^{128}$.
  - Represent same number of values between 2^n and 2^{n+1} as 2^{n+1} and 2^{n+2}

- Thus, floating-point arithmetic has "issues"
  - Small roundoff errors can accumulate with multiplications or exponentiations, resulting in big errors.
  - Rounding errors can invalidate many basic arithmetic principles such as the associative law, $(x + y) + z = x + (y + z)$.
- The IEEE 754 standard guarantees that all machines will produce the same results—but those results may not be mathematically accurate!

Limits of the IEEE representation

- Even some integers cannot be represented in the IEEE format.

```
int x   = 33554431;
float y = 33554431;
printf( "%d\n", x );
printf( "%f\n", y );
```

```
33554431
33554432.000000
```

- Some simple decimal numbers cannot be represented exactly in binary to begin with.

$$0.10_{10} = 0.0001100110011\ldots_2$$

0.10

- During the Gulf War in 1991, a U.S. Patriot missile failed to intercept an Iraqi Scud missile, and 28 Americans were killed.
- A later study determined that the problem was caused by the inaccuracy of the binary representation of 0.10.
  - The Patriot incremented a counter once every 0.10 seconds.
  - It multiplied the counter value by 0.10 to compute the actual time.
- However, the (24-bit) binary representation of 0.10 actually corresponds to 0.099999904632568359375, which is off by 0.000000095367431640625.
- This doesn't seem like much, but after 100 hours the time ends up being off by 0.34 seconds—enough time for a Scud to travel 500 meters!
- Professor Skeel wrote a short article about this.

Roundoff Error and the Patriot Missile. SIAM News, 25(4):11, July 1992.

Floating-point addition example

- To get a feel for floating-point operations, we'll do an addition example.
  - To keep it simple, we'll use base 10 scientific notation.
  - Assume the mantissa has four digits, and the exponent has one digit.
- An example for the addition:

$$99.99 + 0.161 = 100.151$$

- As normalized numbers, the operands would be written as:

$9.999 * 10^{1}$ $1.610 * 10^{-1}$

Steps 1-2: the actual addition

1. **Equalize the exponents.**

   The operand with the smaller exponent should be rewritten by increasing its exponent and shifting the point leftwards.

   $$1.610 * 10^{-1} = 0.01610 * 10^{1}$$

   With four significant digits, this gets rounded to: 0.016

   This can result in a loss of least significant digits—the rightmost 1 in this case. But rewriting the number with the larger exponent could result in loss of the *most* significant digits, which is much worse.

2. **Add the mantissas.**

$$\begin{array}{rrcl} & 9.999 & * & 10^{1} \\ + & 0.016 & * & 10^{1} \\ \hline & 10.015 & * & 10^{1} \end{array}$$

Steps 3-5: representing the result

3. **Normalize the result if necessary.**

   $10.015 * 10^1 = 1.0015 * 10^2$

   This step may cause the point to shift either left or right, and the exponent to either increase or decrease.

4. **Round the number if needed.**

   $1.0015 * 10^2$ gets rounded to $1.002 * 10^2$

5. **Repeat Step 3 if the result is no longer normalized.**

   We don't need this in our example, but it's possible for rounding to add digits—for example, rounding 9.9995 yields 10.000.

Our result is $1.002*10^2$, or 100.2 . The correct answer is 100.151, so we have the right answer to four significant digits, but there's a small error already.

Example

❑ Calculate 0 1000 0001 110…0 plus 0 1000 0010 00110..0 both are single-precision IEEE 754 representation

1. 1st number: $1.11_2 \times 2^{(129-127)}$; 2nd number: $1.0011_2 \times 2^{(130-127)}$
2. Compare the e field: 1000 0001 < 1000 0010
3. Align exponents to 1000 0010; so the 1st number becomes: $0.111_2 \times 2^3$
4. Add mantissa

$$\begin{array}{r} 1.0011 \\ +0.1110 \\ \hline 10.0001 \end{array}$$

5. So the sum is: $10.0001 \times 2^3 = 1.00001 \times 2^4$

So the IEEE 754 format is: 0 1000 0011 000010…0

Multiplication

- To multiply two floating-point values, first multiply their magnitudes and add their exponents.

$$\begin{array}{rrcl} & 9.999 & * & 10^{1} \\ * & 1.610 & * & 10^{-1} \\ \hline & 16.098 & * & 10^{0} \end{array}$$

- You can then round and normalize the result, yielding $1.610 * 10^{1}$.
- The sign of the product is the exclusive-or of the signs of the operands.
  - If two numbers have the same sign, their product is positive.
  - If two numbers have different signs, the product is negative.

$$0 \oplus 0 = 0 \qquad 0 \oplus 1 = 1 \qquad 1 \oplus 0 = 1 \qquad 1 \oplus 1 = 0$$

- This is one of the main advantages of using signed magnitude.

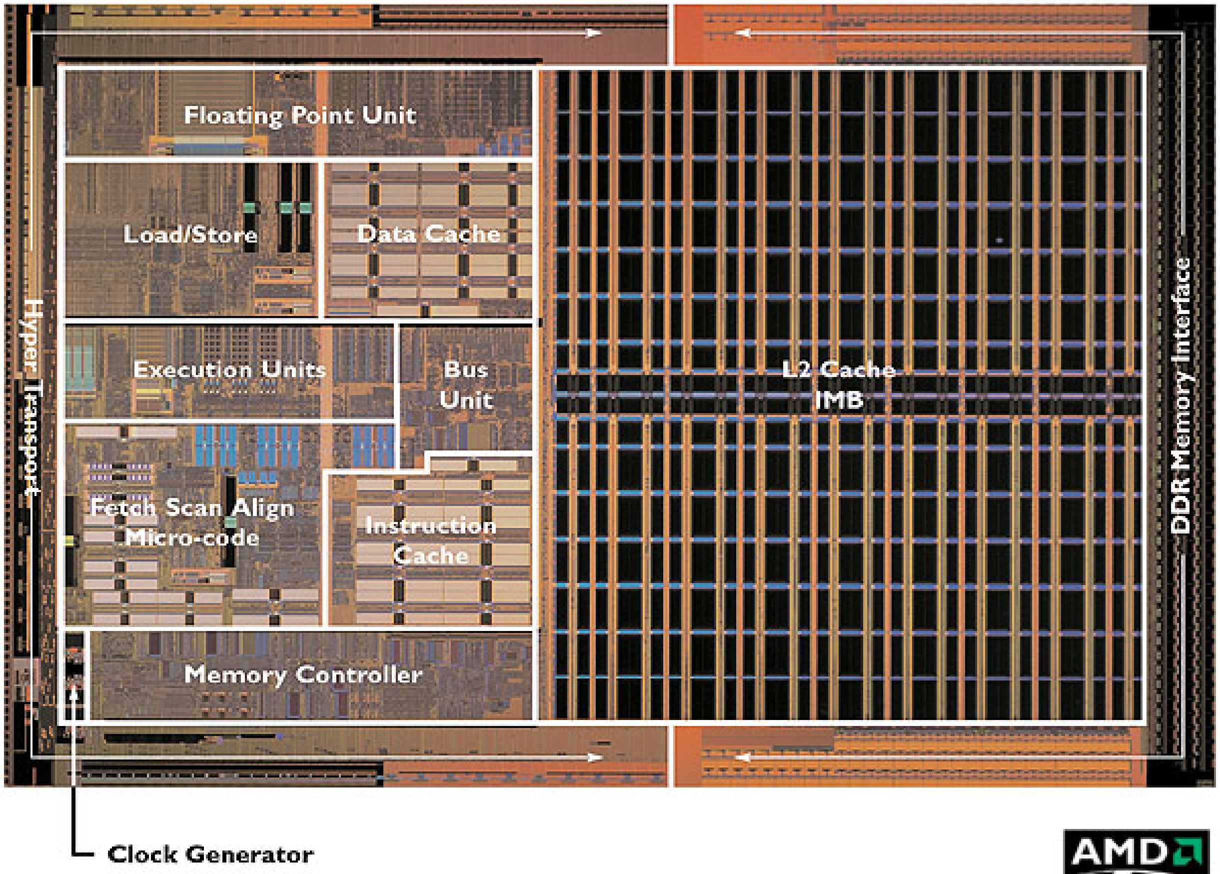
The history of floating-point computation

- **In the past, each machine had its own implementation of floating-point arithmetic hardware and/or software.**
  - **It was impossible to write portable programs that would produce the same results on different systems.**
- **It wasn’t until 1985 that the IEEE 754 standard was adopted.**
  - **Having a standard at least ensures that all compliant machines will produce the same outputs for the same program.**

Floating-point hardware

- When floating point was introduced in microprocessors, there wasn't enough transistors on chip to implement it.
  - You had to buy a floating point co-processor (e.g., the Intel 8087)
- As a result, many ISA's use separate registers for floating point.
- Modern transistor budgets enable floating point to be on chip.
  - Intel's 486 was the first x86 with built-in floating point (1989)
- Even the newest ISA's have separate register files for floating point.
  - Makes sense from a floor-planning perspective.

FPU like co-processor on chip



Summary

- **The IEEE 754 standard defines number representations and operations for floating-point arithmetic.**
- **Having a finite number of bits means we can’t represent all possible real numbers, and errors will occur from approximations.**