CS244, STELLENBOSCH UNIVERSITY

TUT 6: INTEL 32-BIT FLOATING-POINT ARITHMETIC

DUE: 5 NOVEMBER 2021, 23:59

> There is one very good reason to learn programming, but it has nothing to do with preparing for high-tech careers or with making sure one is computer literate in order to avoid being cynically manipulated by the computers of the future. The real value of learning to program can only be understood if we look at learning to program as an exercise of the intellect, as a kind of modern-day Latin that we learn to sharpen our minds.
>
> —Roger C. Schank, *The Cognitive Computer*

1 FLOATING-POINT ARITHMETIC

Not all CPUs support floating-point operations. The first processors within the Intel family (8088/8086, 80286, and 80386) required special processors called coprocessors (8087, 80287 and, 80387) that contained the floating-point unit (FPU). The FPU was first integrated into the 80486 processor.[1] FPUs were expensive and often not an option a typical user would consider when purchasing a PC.

However, floating-point calculations play an important role in computing, whether you need to compute compound interest or calculate the orbit of a satellite. Today, some embedded processors do not possess a FPU, yet still have to process floating-point data. This is usually achieved by emulating floating-point operations. The IEEE 754 standard defines the format for binary floating-point representation of 32-bit single-precision, 64-bit double-precision, and 80-bit extended-precision numbers.

1.1 The IEEE 754 standard for 32-bit floating-point arithmetic

NOTE: As we considered the standard in class, what follows is a brief recap. Refer to Appendix B in your textbook for more details.

1.1.1 Layout

According to the standard, a 32-bit floating-point number has the following layout, as illustrated in the figure below: a 1-bit field for the sign, an 8-bit field for the exponent, and a 23-bit field for the fraction.

[1] And even so, Intel found a way to make money from defective chips: They called the 486 CPU with integrated FPU the 486DX, and if a 486DX chip had a defective FPU (which could happen during the manufacturing process), it was sold as a 486SX, with the FPU disabled, but with the option to buy a separate 487SX as an FPU. But this 487SX was in fact a fully functional 486DX chip (with an extra pin) which just checked that the 486SX was installed, and then disabled it, taking over all of its functions. If you wonder

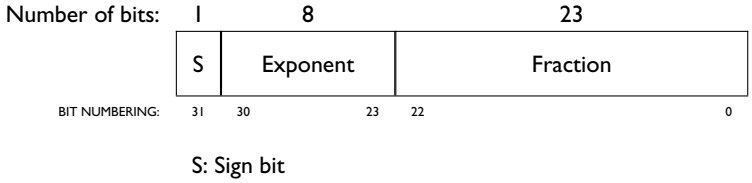

Figure 1: The layout of a 32-bit floating-point number according to the IEEE 754 standard.

The sign bit determines whether the value represented by the real number is positive (sign bit set to 0) or negative (sign bit set to 1). The exponent does not have a sign bit. Instead, it is represented as a positive (unsigned) value and called a *biased exponent*, also called *excess notation*. Subtracting the bias value from the exponent yields its actual value. The bias of the exponent of 32-bit single-precision floating-point numbers is 127. For example, a biased exponent of 129 represents the exponent 2, from $129 - 127$, while a negative exponent such as -3 has a biased representation of 124, from $-3 + 127$.

Floating-point numbers are based on scientific notation. As part of the normalisation process, all fractions are preceded by an implied 1 before the binary point. The literature sometimes refer to this as the J bit or leading bit (or leading 1). It is always present, but never explicitly stored. The value of the J bit may be set to 1 if either the fraction or biased exponent is non-zero.

1.1.2 Addition of 32-bit floating-point numbers by example

Consider the addition of 133.6 and 4.5. The value 4.5 in scientific binary notation is simply $1.001_{\text{E}}10$. But the value 133.6 is more problematic: In scientific decimal notation, it is $1.336_{\text{E}}2$, and its binary representation is $1.0000101\overline{1100}_{\text{E}}111$. From the repeated bits in the fraction, we deduce that 133.6 cannot be represented exactly, and therefore, it is necessary for the floating-point representation to round up or down, or to truncate bits. Mercifully, this is outside the scope of the tutorial, and we are going to let the combination of the C compiler and assembler take care of encoding.

Suffice it then to say that 133.6 will be encoded in binary (scientific notation) as

$$\underbrace{1}_{\text{J bit}}.0000101\,\underbrace{1001100110011010}_{\text{bits 15 to 0}}{}_{\text{E}}111.$$

We motivate this informally by noting that this representation uses 16 of the 23 fraction bits (numbered 15 down to 0) to store the fraction 0.6, and then the two closest numbers—one greater, and the other, smaller—to 0.6 in this encoding is

$$\begin{aligned} 0.1001100110011010_2 &= 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-15} \\ &= \frac{39\,322}{65\,536} = 0.600\,006\,103\,515\,625, \quad \text{and} \qquad (1) \\ 0.1001100110011001_2 &= 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} \\ &= \frac{39\,321}{65\,536} = 0.599\,990\,844\,726\,562\,5. \qquad (2) \end{aligned}$$

Since (1) is closer to 0.6 than (2), the fraction is encoded using (1).

Rewriting each of 133.6 and 4.5 with a biased exponent means the two values are physically stored as follows.

why I think big companies are evil, here you have it. It's nothing new.

	SIGN	BIASED EXPONENT	FRACTION (WITHOUT LEADING 1)
133.6	0	10000110	00001011001100110011010
4.5	0	10000001	00100000000000000000000

Figure 2: The 32-bit IEEE 754 representations of 133.6 and 4.5.

To perform the addition, we expand the stored values as follows; note that we now work with the J bit explicitly, so that the fraction has 24 bits.

	SIGN	EXPONENT	FRACTION (WITH LEADING 1)
133.6	0	00000111	100001011001100110011010
4.5	0	00000010	100100000000000000000000

Figure 3: The expanded 32-bit representations of 133.6 and 4.5.

Next, we equalise the exponents by shifting the representation of 4.5 five positions to the right.

	SIGN	EXPONENT	FRACTION (WITH LEADING 1)
133.6	0	00000111	100001011001100110011010
4.5	0	00000111	000001001000000000000000

Figure 4: The 32-bit representations of 133.6 and 4.5 with both exponents equalised to $7_{10} = 111_2$.

Since both numbers are positive, the fractions can simply be added at this point, yielding 100010100001100110011010 (in 24 bits), which is stored in 32-bit storage as 00000000100010100001100110011010. To store the result as an IEEE 754-compliant number, the normalisation procedure must now ensure that the most-significant 1 is at bit position 24, if necessary by either shifting to the left (and decrementing the exponent) or shifting to the left (and incrementing the exponent). However, for the current example, the most-significat 1 is already at bit position 24, and therefore, shifting is unnecessary.

The result in scientific binary notation is 1.00010100001100110011011E111, which will be stored as an IEEE 754-compliant 32-bit number as follows.

	SIGN	BIASED EXPONENT	FRACTION
138.100006103515625	0	10000110	00010100001100110011010

Figure 5: The 32-bit IEEE 754 representation of the result of 133.6 + 4.5.

2 ALGORITHMS FOR ADDITION

We now consider the algorithms for the addition and normalisation of 32-bit floating-point numbers. Note that that algorithm for addition uses the algorithm for normalisation as a subroutine, but that the implementation does not need to be split into separate subroutines.

Addition of two 32-bit floating-point numbers r_1 and r_2

```
(s1, e1, f1) ← r1                [expand r1 into its sign, exponent, and fraction]
(s2, e2, f2) ← r2                [expand r2 into its sign, exponent, and fraction]
while e1 < e2
    e1 ← e1 + 1
    f1 ← f1/2
while e2 < e1
    e2 ← e2 + 1
    f2 ← f2/2
let e3 be the common exponent
if r1 < 0, then f1 ← −f1         [two's complement negation]
if r2 < 0, then f2 ← −f2         [two's complement negation]
f3 ← f1 + f2
if f3 < 0
    f3 ← −f3                     [two's complement negation]
    set s3 to reflect a negative result
else
    set s3 to reflect a positive result
normalize s3, e3, and f3
combine s3, e3, and f3 to form a 32-bit real value r3
return r3
```

Normalization for a 32-bit floating-point number

```
if the fraction is equal to 0, then stop
while the leftmost part of the fraction (bits 31–24) contains a non-zero bit
    shift the fraction one position to the right
    increment the exponent
while bit 23 is not equal to 1          [bit 23 is the 24th bit from the right]
    shift the fraction one position to the left
    decrement the exponent
```

NOTE: (1) The two algorithms assume that a fraction is stored in a 32-bit register or as a double word in memory. (2) When expanding or combining, you have to handle zero according to the standard (in particular, the J bit), but you do not have to handle the infinities, not-a-number, or denormalised numbers.

3 INSTRUCTIONS

1. Implement addition of 32-bit floating-point numbers in 32-bit Intel assembly language. Your function MUST be callable from a C language program, and MUST have the following prototype:

   ```
   void addf(float a, float b, float *x);
   ```

   Use the parameters `a` and `b` as operands, and return the answer via the pointer parameter `x`.

2. You MUST save your assembly language implementation in a file called `fadd.asm` in the `src` directory of your repository.

3. You only have to implement one function, which is to say, you SHOULD do normalisation in the `addf` function.

4. In addition to the instructions covered for Tutorial 3 to 5, also look at the `neg`, `sets`, and `shld` instructions. *You MUST NOT use any of the Intel floating-point instructions.* Remember to use 8-bit and 16-bit registers where appropriate.

5. If you find it helpful, make an implementation in C first, but such a C implementation does not count for any marks.

Finally, note that in some cases your answers may be reported as being incorrect when compared to the results generated by the FPU. For example, using `fadd` to perform $3.7 + 4.1$ will yield incorrect results. The reason for this is that all floating-point operations are performed using 80-bit extended precision, allowing for more accurate calculations. Furthermore, the rounding mode—which we did not address in this tutorial—also has an effect on the final result. You will not be penalised if your results differ from those of C, *but they must be correct with respect to the algorithms given above.*

REPOSITORY Your repository is available at:

`rw244-2021@gitolite.cs.sun.ac.za:`⟨*US number*⟩`/tut6`

It includes a makefile.

MARKING Marking will be by test cases only. There are no marks for style, and no caps for crashing or memory use.

PLAGIARISM You have to complete the plagiarism declaration in your repository. The signing script is included in the `sub` directory.