7

Floating-Point Unit

CHAPTER 7
FLOATING-POINT UNIT

The Intel Architecture Floating-Point Unit (FPU) provides high-performance floating-point processing capabilities. It supports the real, integer, and BCD-integer data types and the floating-point processing algorithms and exception handling architecture defined in the IEEE 754 and 854 Standards for Floating-Point Arithmetic. The FPU executes instructions from the processor's normal instruction stream and greatly improves the efficiency of Intel Architecture processors in handling the types of high-precision floating-point processing operations commonly found in scientific, engineering, and business applications.

This chapter describes the data types that the FPU operates on, the FPU's execution environment, and the FPU-specific instruction set. Detailed descriptions of the FPU instructions are given in Chapter 3, *Instruction Set Reference*, in the *Intel Architecture Software Developer's Manual, Volume 2*.

7.1. COMPATIBILITY AND EASE OF USE OF THE INTEL ARCHITECTURE FPU

The architecture of the Intel Architecture FPU has evolved in parallel with the architecture of early Intel Architecture processors. The first Intel Math Coprocessors (the Intel 8087, Intel 287, and Intel 387) were companion processors to the Intel 8086/8088, Intel 286, and Intel386 processors, respectively, and were designed to improve and extend the numeric processing capability of the Intel Architecture. The Intel486 DX processor for the first time integrated the CPU and the FPU architectures on one chip. The Pentium processor's FPU offered the same architecture as the Intel486 DX processor's FPU, but with improved performance. The Pentium Pro processor's FPU further extended the floating-point processing capability of Intel Architecture family of processors and added several new instructions to improve processing throughput.

Throughout this evolution, compatibility among the various generations of FPUs and math coprocessors has been maintained. For example, the Pentium Pro processor's FPU is fully compatible with the Pentium and Intel486 DX processors's FPUs.

Each generation of the Intel Architecture FPUs have been explicitly designed to deliver stable, accurate results when programmed using straightforward "pencil and paper" algorithms, bringing the functionality and power of accurate numeric computation into the hands of the general user. The IEEE 754 standard specifically addresses this issue, recognizing the fundamental importance of making numeric computations both easy and safe to use.

For example, some processors can overflow when two single-precision floating-point numbers are multiplied together and then divided by a third, even if the final result is a perfectly valid 32-bit number. The Intel Architecture FPUs deliver the correctly rounded result. Other typical examples of undesirable machine behavior in straightforward calculations occur when computing financial rate of return, which involves the expression $(1 + i)^n$ or when solving for roots of a quadratic equation:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If *a* does not equal 0, the formula is numerically unstable when the roots are nearly coincident or when their magnitudes are wildly different. The formula is also vulnerable to spurious over/underflows when the coefficients *a*, *b*, and *c* are all very big or all very tiny. When single-precision (4-byte) floating-point coefficients are given as data and the formula is evaluated in the FPU's normal way, keeping all intermediate results in its stack, the FPU produces impeccable single-precision roots. This happens because, by default and with no effort on the programmer's part, the FPU evaluates all those sub-expressions with so much extra precision and range as to overwhelm almost any threat to numerical integrity.

If double-precision data and results were at issue, a better formula would have to be used, and once again the FPU's default evaluation of that formula would provide substantially enhanced numerical integrity over mere double-precision evaluation.

On most machines, straightforward algorithms will not deliver consistently correct results (and will not indicate when they are incorrect). To obtain correct results on traditional machines under all conditions usually requires sophisticated numerical techniques that go beyond typical programming practice. General application programmers using straightforward algorithms will produce much more reliable programs using the Intel architectures. This simple fact greatly reduces the software investment required to develop safe, accurate computation-based products.

Beyond traditional numeric support for scientific applications, the Intel architectures have built-in facilities for commercial computing. They can process decimal numbers of up to 18 digits without round-off errors, performing **exact arithmetic** on integers as large as 2^{64} (or 10^{18}). Exact arithmetic is vital in accounting applications where rounding errors may introduce monetary losses that cannot be reconciled.

The Intel FPU's contain a number of optional numerical facilities that can be invoked by sophisticated users. These advanced features include directed rounding, gradual underflow, and programmed exception-handling facilities.

These automatic exception-handling facilities permit a high degree of flexibility in numeric processing software, without burdening the programmer. While performing numeric calculations, the processor automatically detects exception conditions that can potentially damage a calculation (for example, $X \div 0$ or $\sqrt{X}$ when $X < 0$). By default, on-chip exception logic handles these exceptions so that a reasonable result is produced and execution may proceed without program interruption. Alternatively, the processor can invoke a software exception handler to provide special results whenever various types of exceptions are detected.

7.2. REAL NUMBERS AND FLOATING-POINT FORMATS

This section describes how real numbers are represented in floating-point format in the Intel Architecture FPU. It also introduces terms such as normalized numbers, denormalized numbers, biased exponents, signed zeros, and NaNs. Readers who are already familiar with floating-point processing techniques and the IEEE standards may wish to skip this section.

7.2.1. Real Number System

As shown in Figure 7-1, the real-number system comprises the continuum of real numbers from minus infinity (–∞) to plus infinity (+∞).

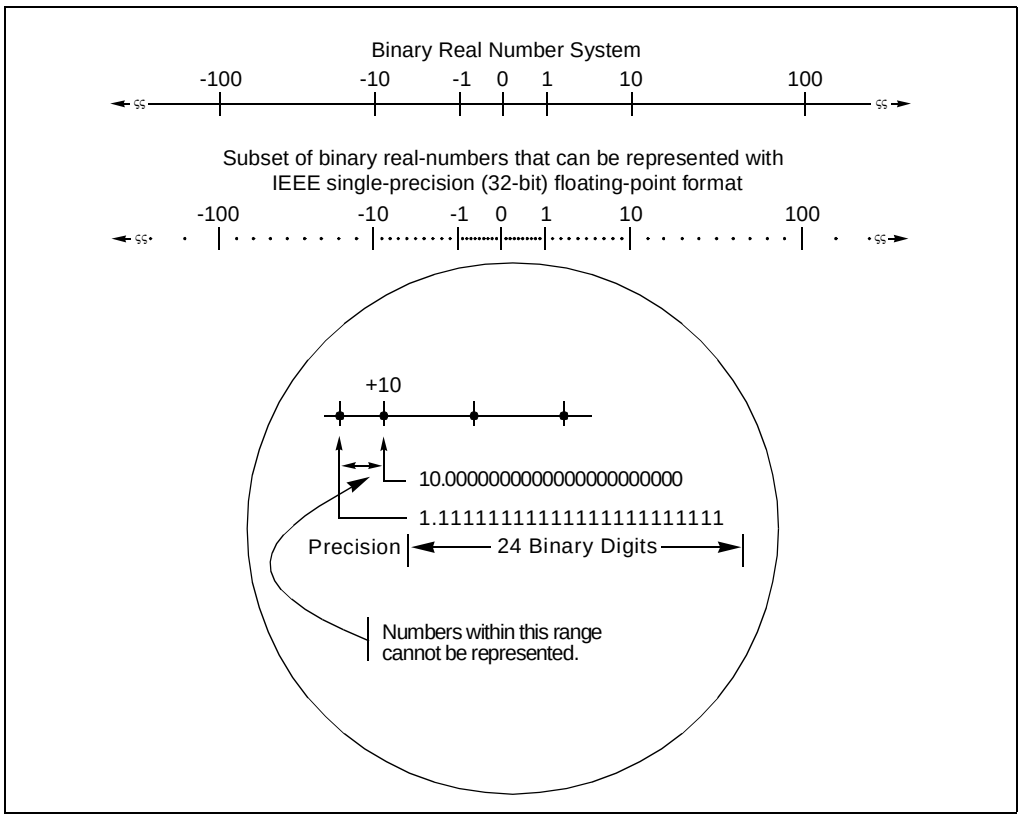

Figure 7-1. Binary Real Number System

Because the size and number of registers that any computer can have is limited, only a subset of the real-number continuum can be used in real-number calculations. As shown at the bottom of Figure 7-1, the subset of real numbers that a particular FPU supports represents an approximation of the real number system. The range and precision of this real-number subset is determined by the format that the FPU uses to represent real numbers.

7.2.2. Floating-Point Format

To increase the speed and efficiency of real-number computations, computers or FPUs typically represent real numbers in a binary floating-point format. In this format, a real number has three

parts: a sign, a significand, and an exponent. Figure 7-2 shows the binary floating-point format that the Intel Architecture FPU uses. This format conforms to the IEEE standard.

The sign is a binary value that indicates whether the number is positive (0) or negative (1). The significand has two parts: a 1-bit binary integer (also referred to as the J-bit) and a binary fraction. The J-bit is often not represented, but instead is an implied value. The exponent is a binary integer that represents the base-2 power that the significand is raised to.

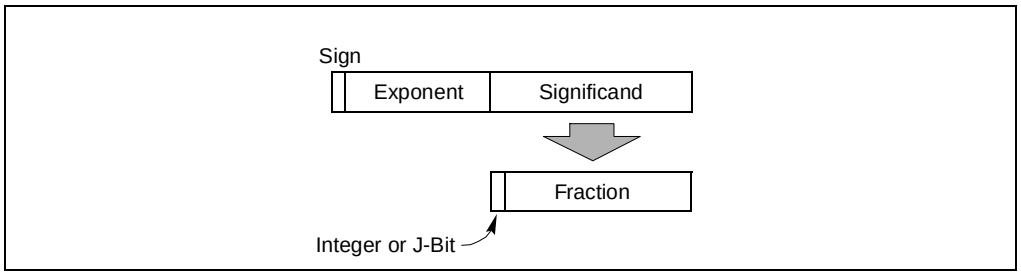

Figure 7-2. Binary Floating-Point Format

Table 7-1 shows how the real number 178.125 (in ordinary decimal format) is stored in floating-point format. The table lists a progression of real number notations that leads to the single-real, 32-bit floating-point format (which is one of the floating-point formats that the FPU supports). In this format, the significand is normalized (see Section 7.2.2.1., “Normalized Numbers”) and the exponent is biased (see Section 7.2.2.2., “Biased Exponent”). For the single-real format, the biasing constant is +127.

7.2.2.1. NORMALIZED NUMBERS

In most cases, the FPU represents real numbers in normalized form. This means that except for zero, the significand is always made up of an integer of 1 and the following fraction:

1.fff...ff

For values less than 1, leading zeros are eliminated. (For each leading zero eliminated, the exponent is decremented by one.)

Table 7-1. Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	$1.78125E_{10}2$		
Scientific Binary	$1.0110010001E_{2}111$		
Scientific Binary (Biased Exponent)	$1.0110010001E_{2}10000110$		
Single-Real Format	Sign	Biased Exponent	Normalized Significand
	0	10000110	01100100010000000000000 1. (Implied)

Representing numbers in normalized form maximizes the number of significant digits that can be accommodated in a significand of a given width. To summarize, a normalized real number consists of a normalized significand that represents a real number between 1 and 2 and an exponent that specifies the number's binary point.

7.2.2.2. BIASED EXPONENT

The FPU represents exponents in a biased form. This means that a constant is added to the actual exponent so that the biased exponent is always a positive number. The value of the biasing constant depends on the number of bits available for representing exponents in the floating-point format being used. The biasing constant is chosen so that the smallest normalized number can be reciprocated without overflow.

(See Section 7.4.1., "Real Numbers" for a list of the biasing constants that the FPU uses for the various sizes of real data-types.)

7.2.3. Real Number and Non-number Encodings

A variety of real numbers and special values can be encoded in the FPU's floating-point format. These numbers and values are generally divided into the following classes:

- Signed zeros.
- Denormalized finite numbers.
- Normalized finite numbers.
- Signed infinities.
- NaNs.
- Indefinite numbers.

(The term NaN stands for "Not a Number.")

Figure 7-3 shows how the encodings for these numbers and non-numbers fit into the real number continuum. The encodings shown here are for the IEEE single-precision (32-bit) format, where the term "S" indicates the sign bit, "E" the biased exponent, and "F" the fraction. (The exponent values are given in decimal.)

The FPU can operate on and/or return any of these values, depending on the type of computation being performed. The following sections describe these number and non-number classes.

7.2.3.1. SIGNED ZEROS

Zero can be represented as a +0 or a –0 depending on the sign bit. Both encodings are equal in value. The sign of a zero result depends on the operation being performed and the rounding mode being used. Signed zeros have been provided to aid in implementing interval arithmetic. The sign of a zero may indicate the direction from which underflow occurred, or it may indicate the sign of an ∞ that has been reciprocated.

7.2.3.2. NORMALIZED AND DENORMALIZED FINITE NUMBERS

Non-zero, finite numbers are divided into two classes: normalized and denormalized. The normalized finite numbers comprise all the non-zero finite values that can be encoded in a normalized real number format between zero and ∞. In the single-real format shown in Figure 7-3, this group of numbers includes all the numbers with biased exponents ranging from 1 to 254_{10} (unbiased, the exponent range is from -126_{10} to $+127_{10}$).

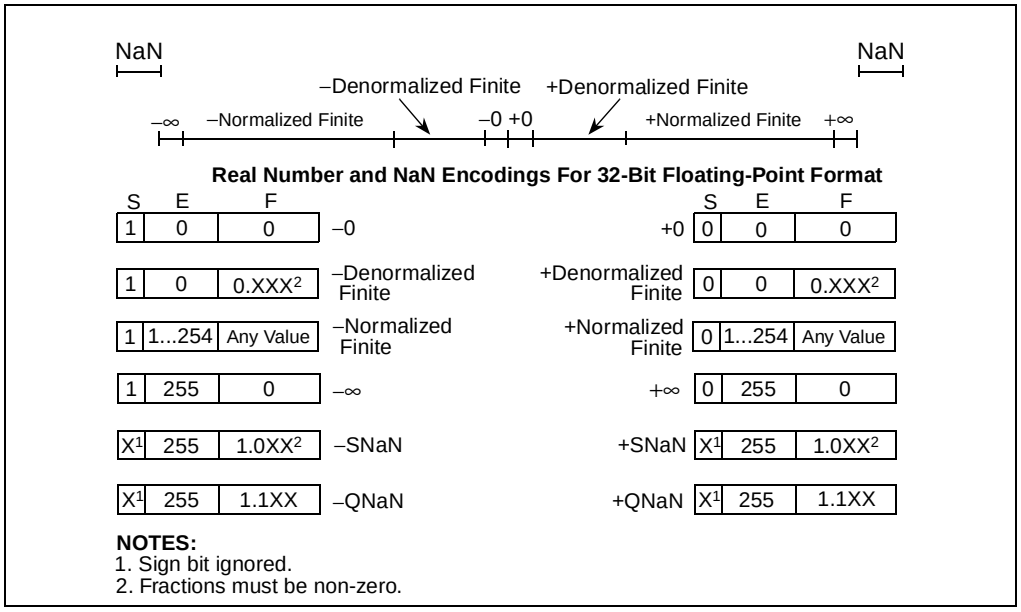

Figure 7-3. Real Numbers and NaNs

When real numbers become very close to zero, the normalized-number format can no longer be used to represent the numbers. This is because the range of the exponent is not large enough to compensate for shifting the binary point to the right to eliminate leading zeros.

When the biased exponent is zero, smaller numbers can only be represented by making the integer bit (and perhaps other leading bits) of the significand zero. The numbers in this range are called **denormalized** (or **tiny**) numbers. The use of leading zeros with denormalized numbers allows smaller numbers to be represented. However, this denormalization causes a loss of precision (the number of significant bits in the fraction is reduced by the leading zeros).

When performing normalized floating-point computations, an FPU normally operates on normalized numbers and produces normalized numbers as results. Denormalized numbers represent an **underflow** condition.

A denormalized number is computed through a technique called gradual underflow. Table 7-2 gives an example of gradual underflow in the denormalization process. Here the single-real format is being used, so the minimum exponent (unbiased) is -126_{10}. The true result in this example requires an exponent of -129_{10} in order to have a normalized number. Since -129_{10} is beyond the allowable exponent range, the result is denormalized by inserting leading zeros until the minimum exponent of -126_{10} is reached.

Table 7-2. Denormalization Process

Operation	Sign	Exponent*	Significand
True Result	0	–129	1.01011100000...00
Denormalize	0	–128	0.10101110000...00
Denormalize	0	–127	0.01010111000...00
Denormalize	0	–126	0.00101011100...00
Denormal Result	0	–126	0.00101011100...00

NOTE:

* Expressed as an unbiased, decimal number.

In the extreme case, all the significant bits are shifted out to the right by leading zeros, creating a zero result.

The FPU deals with denormal values in the following ways:

- It avoids creating denormals by normalizing numbers whenever possible.
- It provides the floating-point underflow exception to permit programmers to detect cases when denormals are created.
- It provides the floating-point denormal-operand exception to permit procedures or programs to detect when denormals are being used as source operands for computations.

When a denormal number in single- or double-real format is used as a source operand and the denormal exception is masked, the FPU automatically **normalizes** the number when it is converted to extended-real format.

7.2.3.3. SIGNED INFINITIES

The two infinities, $+\infty$ and $-\infty$, represent the maximum positive and negative real numbers, respectively, that can be represented in the floating-point format. Infinity is always represented by a zero significand (fraction and integer bit) and the maximum biased exponent allowed in the specified format (for example, 255_{10} for the single-real format).

The signs of infinities are observed, and comparisons are possible. Infinities are always interpreted in the affine sense; that is, $-\infty$ is less than any finite number and $+\infty$ is greater than any finite number. Arithmetic on infinities is always exact. Exceptions are generated only when the use of an infinity as a source operand constitutes an invalid operation.

Whereas denormalized numbers represent an underflow condition, the two infinity numbers represent the result of an overflow condition. Here, the normalized result of a computation has a biased exponent greater than the largest allowable exponent for the selected result format.

7.2.3.4. NANS

Since NaNs are non-numbers, they are not part of the real number line. In Figure 7-3, the encoding space for NaNs in the FPU floating-point formats is shown above the ends of the real number line. This space includes any value with the maximum allowable biased exponent and a non-zero fraction. (The sign bit is ignored for NaNs.)

The IEEE standard defines two classes of NaN: quiet NaNs (QNaNs) and signaling NaNs (SNaNs). A QNaN is a NaN with the most significant fraction bit set; an SNaN is a NaN with the most significant fraction bit clear. QNaNs are allowed to propagate through most arithmetic operations without signaling an exception. SNaNs generally signal an invalid-operation exception whenever they appear as operands in arithmetic operations. Exceptions are discussed in Section 7.7., “Floating-Point Exception Handling”.

See Section 7.6., “Operating on NaNs”, for detailed information on how the FPU handles NaNs.

7.2.4. Indefinite

For each FPU data type, one unique encoding is reserved for representing the special value **indefinite**. For example, when operating on real values, the real indefinite value is a QNaN (see Section 7.4.1., “Real Numbers”). The FPU produces indefinite values as responses to masked floating-point exceptions.

7.3. FPU ARCHITECTURE

From an abstract, architectural view, the FPU is a coprocessor that operates in parallel with the processor’s integer unit (see Figure 7-4). The FPU gets its instructions from the same instruction decoder and sequencer as the integer unit and shares the system bus with the integer unit. Other than these connections, the integer unit and FPU operate independently and in parallel. (The actual microarchitecture of an Intel Architecture processor varies among the various families of processors. For example, the Pentium Pro processor has two integer units and two FPUs; whereas, the Pentium processor has two integer units and one FPU, and the Intel486 processor has one integer unit and one FPU.)