Managing parallel execution units in a superscalar microprocessor is a complex task, because the microprocessor wants to execute instructions as fast as they can be fetched—yet it must do so in a manner consistent with the instructions' serial interdependencies. These dependencies can become more complicated to resolve when superscalar and superpipelining techniques are combined to create a microprocessor with multiple execution units, each of which is implemented with a deep pipeline. In such chips, the instruction decode logic handles the complex task of examining the pipelines of the execution units to determine when the next instruction is free of dependencies, allowing it to begin execution.

Related to superpipelining and superscalar methods are the techniques of branch prediction, speculative execution, and instruction reordering. Deep pipelines are subject to performance-degrading flushes each time a branch instruction comes along. To reduce the frequency of pipeline flushes due to branch instructions, some microprocessors incorporate branch prediction logic that attempts to make a preliminary guess as to whether the branch will be taken. These guesses are made based on the history of previous branches. The exact algorithms that perform branch prediction vary by implementation and are not always disclosed by the manufacturer, to protect their trade secrets. When the branch prediction logic makes its guess, the instruction fetch and decode logic can speculatively execute the instruction stream that corresponds to the predicted branch result. If the prediction logic is correct, a costly pipeline flush is avoided. If the prediction is wrong, performance will temporarily degrade until the pipeline can be restarted. Hopefully, a given branch prediction algorithm improves performance rather than degrading it by having a worse record than would exist with no prediction at all!

The problem with branch prediction is that it is sometimes wrong, and the microprocessor must back out of any state changes that have resulted from an incorrectly predicted branch. Speculative execution can be taken a step farther in an attempt to eliminate the penalty of a wrong branch prediction by executing both possible branch results. To do this, a superscalar architecture is needed that has enough execution units to speculatively execute extra instructions whose results may not be used. It is a foregone conclusion that one of the branch results will not be valid. There is substantial complexity involved in such an approach because of the duplicate hardware that must be managed and the need to rapidly swap to the correct instruction stream that is already in progress when the result of a branch is finally known.

A superscalar microprocessor will not always be able to keep each of its execution units busy, because of dependencies across sequential instructions. In such a case, the next instruction to be pushed into the execution pipeline must be held until an in-progress instruction completes. Instruction reordering logic reduces the penalty of such instruction stalls by attempting to execute instructions outside the order in which they appear in the program. The microprocessor can prefetch a set of instructions ahead of those currently executing, enabling it to look ahead in the sequence and determine whether a later instruction can be safely executed without changing the behavior of the instruction stream. For such reordering to occur, an instruction must not have any dependencies on those that are being temporarily skipped over. Such dependencies include not only operands but branch possibilities as well. Reordering can occur in a situation in which the ALUs are busy calculating results that are to be used by the next instruction in the sequence, and their latencies are preventing the next instruction from being issued. A load operation that is immediately behind the stalled instruction can be executed out of order if it does not operate on any registers that are being used by the instructions ahead of it. Such reordering boosts throughput by taking advantage of otherwise idle execution cycles.

All of the aforementioned throughput improvement techniques come at a cost of increased design complexity and cost. However, it has been widely noted that the cost of a transistor on an IC is asymptotically approaching zero as tens of millions of transistors are squeezed onto chips that cost only several hundred dollars. Once designed, the cost of implementing deep pipelines, multiple execution units, and the complex logic that coordinates the actions of both continues to decrease over time.

7.6 FLOATING-POINT ARITHMETIC

Conventional arithmetic logic units operate on signed and unsigned integer quantities. Integers suffice for many applications, including loop count variables and memory addresses. However, our world is inherently analog and is best represented by real numbers as compared to discrete integers. Floating-point arithmetic enables the representation and manipulation of real numbers of arbitrary magnitude and precision. Historically, floating-point math was pertinent only to members of the scientific community who regularly perform calculations on large data sets to model many types of natural phenomena. Almost every area of scientific research has benefited from computational analysis, including aerodynamics, geology, medicine, and meteorology. More recently, floating-point math has become more applicable to the mainstream community in such applications as video games that render realistic three-dimensional scenes in real-time as game characters move around in virtual environments.

General mathematics represents numbers of arbitrary magnitude and precision using *scientific notation*, consisting of a signed *mantissa* multiplied by an integer power of ten. The mantissa is greater than or equal to one and less than ten. In other words, the decimal point of the mantissa is shifted left or right until a single digit remains in the 1s column. The number 456.8 would be represented as 4.568×10^2 in scientific notation. All significant digits other than the first one are located to the right of the decimal point. The number –0.000089 has only two significant digits and is represented as -8.9×10^{-5}. Scientific notation enables succinct and accurate representation of very large and very small numbers.

Floating-point arithmetic on a computer uses a format very similar to scientific notation, but binary is used in place of decimal representation. The Institute of Electrical and Electronics Engineers (IEEE) has standardized floating-point representation in several formats to express numbers of increasing magnitude and precision. These formats are used by most hardware and software implementations of floating-point arithmetic for the sake of compatibility and consistency. Figure 7.9 shows the general structure of an IEEE floating point number.

The most significant bit is defined as a sign bit where zero is positive and one is negative. The sign bit is followed by an n-bit exponent with values from 1 to $2^n - 2$ (the minimum and maximum values for the exponent field are not supported for normal numbers). The exponent represents powers of two and can represent negative exponents by means of an exponent *bias*. The bias is a fixed, standardized value that is subtracted from the actual exponent field to yield the true exponent value. It is generally $2^{(n-1)} - 1$. Following the exponent is the binary *significand*, which is a mantissa or modified mantissa. Similar to scientific notation, the mantissa is a number greater than or equal to 1 and less than the radix (2, in this case). Therefore, the whole number portion of the binary mantissa must be 1. Some IEEE floating-point formats hide this known bit and use a modified mantissa to provide an additional bit of precision in the fractional portion of the mantissa. Table 7.5 lists the basic parameters of the four commonly used floating-point formats. The IEEE-754 standard defines several formats including single and double precision. The extended and quadruple precision formats are not explicitly mentioned in the standard, but they are legal derivations from formats that provide for increased precision and exponent ranges.

It is best to use a single-precision example to see how floating-point representation actually works. The decimal number 25.25 is first converted to its binary equivalent: 11001.01. The mantissa and exponent are found by shifting the binary point four places to the left to yield 1.100101×2^4. Us-

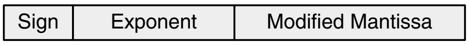

FIGURE 7.9 General IEEE floating-point structure.