CEG 3156: Computer Systems Design (Winter 2024)
Prof. Rami Abielmona
Laboratory #1: Floating-Point Multiplication

January 8, 2024

1 Objective

The objective of this laboratory is to design and build a floating-point multiplier in VHDL.

Upon completion, the student must be able to:

- Design, realize and test a floating-point adder unit;
- Design, realize and test a floating-point multiplier unit;
- Demonstrate a complete understanding for floating-point arithmetic.

2 Pre-Lab

Usually, a pre-lab would be associated with each laboratory, but due to time constraints, this experiment will not require a pre-lab. Successive experiments, however, will have a pre-lab section that has to be completed by the group before attending the allocated laboratory hours.

3 Introduction to Floating-Point Numbers

Dealing with fixed-point arithmetic will limit the usability of a processor. If operations on numbers with fractions (e.g. 10.2445), very small numbers (e.g. 0.000004), or very large numbers (e.g. 42.243x10^5) are required, then a different representation is in order: Enter floating-point arithmetic. The latter term is utilized as the binary point is not fixed, as is the case in integer (fixed-point) arithmetic.

In order to get some of the terminology out of the way, let us discuss a simple floating-point number, such as -2.42x10^3. The '-' symbol indicates the **sign component** of the number, while the '242' indicate the **significant digits component** of the number, and finally the '3' indicates the **scale factor component** of the number. It is interesting to note that the string of significant digits is technically termed the *mantissa* of the number, while the scale factor is appropriately called the *exponent* of the number.

3.1 IEEE Standard for Floating-Point Numbers

Instead of having a myriad of floating-point representations, it was decided that a standard must be conformed to. This standard happens to be the IEEE standard for floating-point number representations. The general form of the representation is the following:

$$(-1)^S * M * 2^E, where \tag{1}$$

S represents the sign bit,
M represents the mantissa and
E represents the exponent

In today's computers, 32 bits is a standard word length, and thus, the IEEE decided that a 32-bit representation of floating-point numbers could benefit from today's bus transfer characteristics. Hence, the *single-precision number* representation (refer to 1) was made out to be 32-bits long, while the *double-precision number* representation (refer to 2) was made out to be 64-bits long. The single-precision number has three main fields:

Sign 1-bit wide, and used to denote the sign of the number (0 signifies + and 1 signifies -);

Exponent 8-bit signed exponent in excess-127 representation (more on this later);

Mantissa 23-bit fractional component.

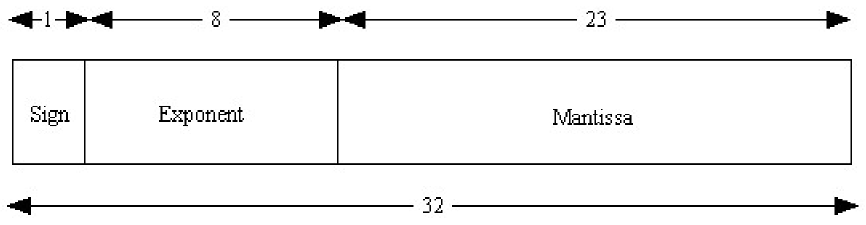

Figure 1: Single-precision floating-point number representation

The 24-bit mantissa (the leading one is implicit) can approximately represent a 7-digit decimal number, while an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. Thus, a total of 32 bits is needed for single-precision number representation, which provides a scale factor of 2^{-126} to 2^{127}. In the case of double-precision, the exponent field grows to 11 bits, while the mantissa field grows to 52 bits, allowing for a scale factor of 2^{-1022} to 2^{1023}, while providing a precision equivalent to about 16 decimal digits.

The excess-127 representation mentioned when discussing the exponent portion above, is utilized to efficiently compare the relative sizes of two floating-point numbers. Instead of storing the exponent (E) as a signed number, we store its unsigned integer representation ($E' = E + 127$). This gives us a range for E' of $0 \leq E' \leq 255$. While the 0 and 255 end values are used to represent special numbers (exact 0, infinity and denormal numbers), the operating range of E' becomes $1 \leq E' \leq 254$, thus, limiting the range of E to $-126 \leq E \leq 127$. (Note that if working with double-precision numbers, an excess-1023 representation is utilized).

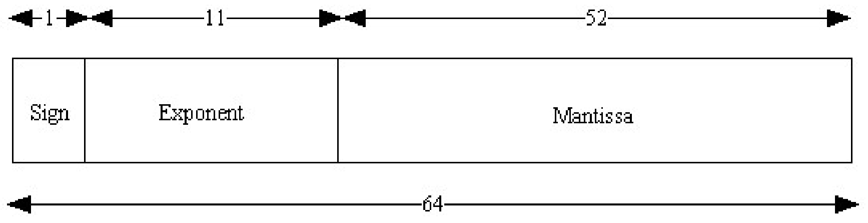

Figure 2: Double-precision floating-point number representation

A final note about the IEEE standard is that the mantissa component is always **normalized**. The latter implies that the decimal point is placed to the right of the first (nonzero) significant digit. Hence, the 23 bits stored in the M field actually represent the fractional part of the mantissa, that is, the bits to the right of the binary point. As aforementioned, the most significant bit of the mantissa is always equal to 1, due to binary normalization.

Revisiting our original formula representation, let us now formalize the 32-bit single-precision floating-point number representation as:

$$(-1)^S * (1 + M) * 2^{E-127}, where \tag{2}$$

S represents the sign (1-bit),
M represents the mantissa (23-bit) and
E represents the exponent (8-bit)

3.2 A Simple Example

Let us try and represent the decimal number $(-0.75)_{10}$ in IEEE floating-point format. First off, we notice that $(-0.75)_{10} = (-3/4)_{10} = (-3/2^2)_{10}$. In binary notation, we have $(-0.11)_2 = (-0.11)_2$ x $2^0 = (-1.1)_2$ x 2^{-1}.

Referring to equation (2), we can represent our number as:

$$(-1)^1 * (1 + .10000000000000000000000_2) * 2^{126-127} \tag{3}$$

Thus, our single-precision representation of the number is $(10111111010000000000000000000000)_2$, where

- The sign bit is $(1)_2$, for negative numbers;
- The exponent is $(01111110)_2$, to represent $(126)_{10}$;
- The mantissa is $(10000000000000000000000)_2$, to represent the fractional part $(.1)_2$.

3.3 Floating-Point Addition

In order to perform floating-point addition, a simple algorithm is realized:

1. Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents;
2. Set the exponent of the result equal to the larger exponent;
3. Perform addition on the mantissas and determine the sign of the result;
4. Normalize the resulting value, if necessary.

Refer to figure 4 for a flow diagram of the floating-point addition operation. The figure is extracted from our course textbook, so please refer to section 4.8 of the book for a detailed explanation of the interworking of the flow chart, as well as floating-point addition.

Also shown in this document is the block diagram implementation of a floating-point adder (figure 5). This diagram indicates the data path of the addition, and abstracts away the control path of the unit.

3.4 Floating-Point Multiplication

In order to perform floating-point multiplication, a simple algorithm is realized:

1. Add the exponents and subtract 127;
2. Multiply the mantissas and determine the sign of the result;
3. Normalize the resulting value, if necessary.

Refer to figure 6 for a flow diagram of the floating-point multiplication operation. The figure is extracted from our course textbook, so please refer to section 4.8 of the book for a detailed explanation of the interworking of the flow chart, as well as floating-point multiplication.

4 Laboratory

In this lab, you will proceed to first implement a floating-point adder as shown in figure 5, and design and implement a floating-point multiplier as specified in figure 6. The input/output specification of both the floating-point adder and multiplier is shown in table 1. We will be working with **7-bit exponent widths** and **8-bit mantissa widths**, while the sign bit obviously remains at 1-bit wide. Also note that the 7-bit exponent will be represented in **excess-63 format**.

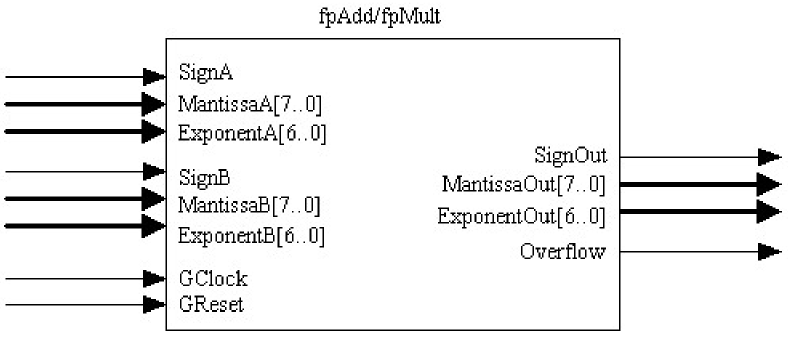

Figure 3: Floating-point adder/multiplier entity

Port Type	*Name*	*Description*
Input	GClock	Global clock needed to synchronize the circuitry
Input	GReset	Global reset needed to bring the internals to known states
Input	SignA	Sign bit for the A input: 0 = positive, 1 = negative
Input	MantissaA[7..0]	Mantissa for the A input
Input	ExponentA[6..0]	Exponent for the A input
Input	SignB	Sign bit for the B input: 0 = positive, 1 = negative
Input	MantissaB[7..0]	Mantissa for the B input
Input	ExponentB[6..0]	Exponent for the B input
Output	SignOut	Sign bit for the output 0 = positive, 1 = negative
Output	MantissaOut[7..0]	Mantissa for the output
Output	ExponentOut[6..0]	Exponent for the output
Output	Overflow	Overflow used to indicate whether overflow has been detected

Table 1: Input/Output Specification

5 Design Restrictions

- Verilog implementations will not be accepted. Perform all implementations in VHDL code only
- Behavioral level of modeling will not be accepted. Design should be done at the structural level of modeling
- Register Transfer Logic (RTL) design and coding will is mandatory
- Use graphical design for the top-level entity, and use your judgment for any other sub-blocks. However, all atomic modules have to be implemented in VHDL (i.e. D flip-flop, 1-bit adder, 1-bit comparator and so on)
- This lab involves 16-bit floating-point arithmetic, hence, students must design and test their modules using this format, and not the single- or double-precision formats standardized by the IEEE
- No core instantiations are allowed (i.e. LPMs from Altera or free IP cores from the Internet). All building blocks have to be designed and realized by the group
- The top level entity is given in input/output specification format, but the internals are left up to the group. A sample schematic solution for the floating-point adder has been given, but the group is free to design the floating-point adder and multiplier using another methodology
- The design has to be synchronous and globally reset-able. This means that global clock and reset signals are required in both functional blocks (the fp_adder and the fp_multiplier)
- Simulate both designs and check your simulation results with your theoretical ones (e.g. add and multiply the following two numbers: 10.1 and 6.5)
- Download the design to the Cyclone chip on your DE-2 boards, and use the push-buttons to input the two floating-point numbers, and sixteen LEDs to demonstrate correct output functionality
- Indicate the detection of overflow by using the decimal point on the BCD 7-segment decoders of the Cyclone FPGA
- Each group must demonstrate a working version of the laboratory to the TA, before the due date of the report

6 Report Reminders

- Include timing simulations with explanation for all VHDL source files
- Describe and comment all your VHDL source files

- Include a flowchart representation of your solution to the problem
- Include a block diagram of your solution to the problem
- If using ASM design, include all appropriate charts and paths (control and data)
- If using FSM design, include the state diagram with all appropriate inputs and outputs
- Describe, in your own words, your solution to the problem
- Describe your design obstacles and how they were overcome
- Append all VHDL source code and graphical design files to your report
- Submit a soft copy of all VHDL and graphical design files with your report

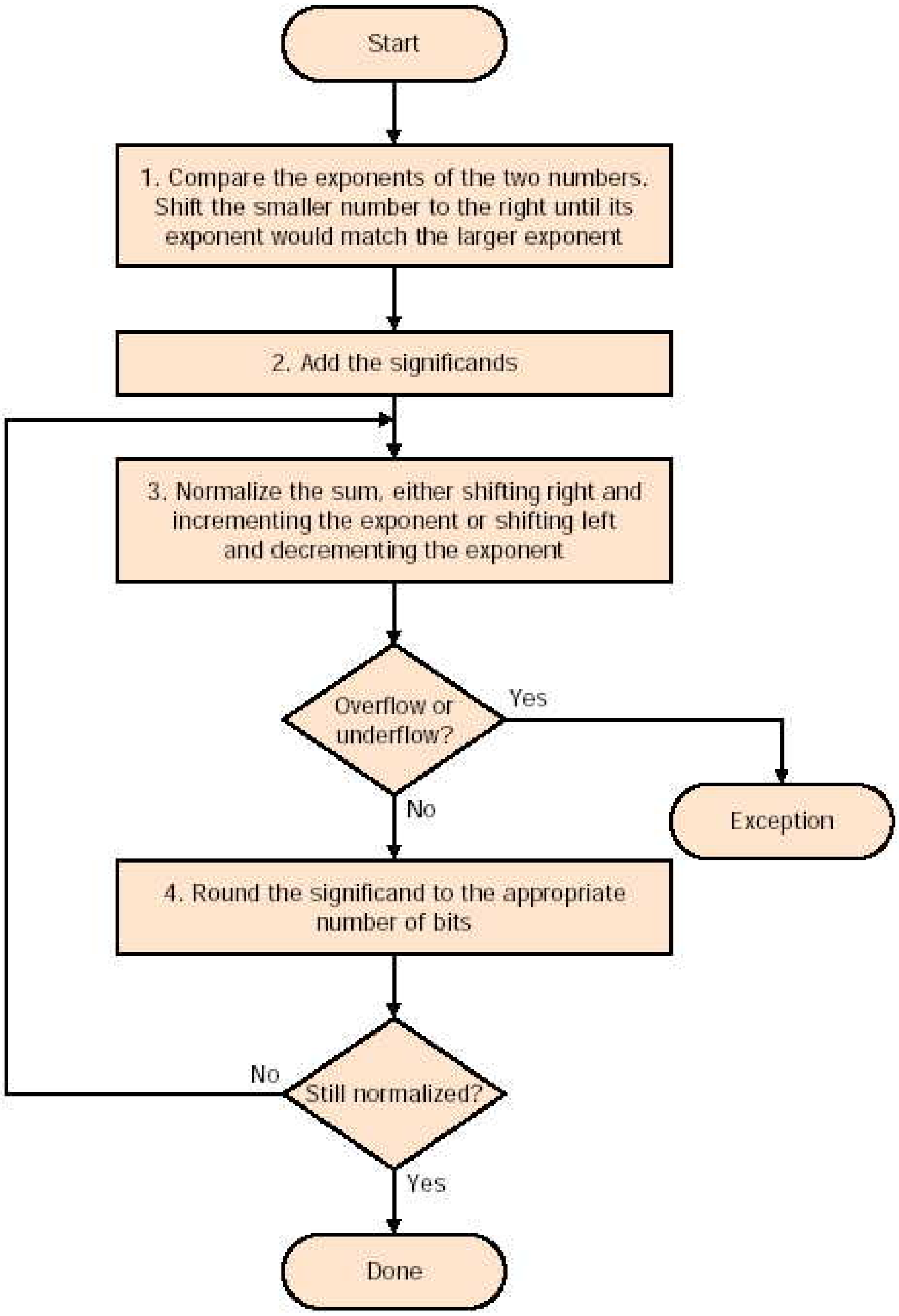
Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

Figure 4: Floating-point adder flowchart

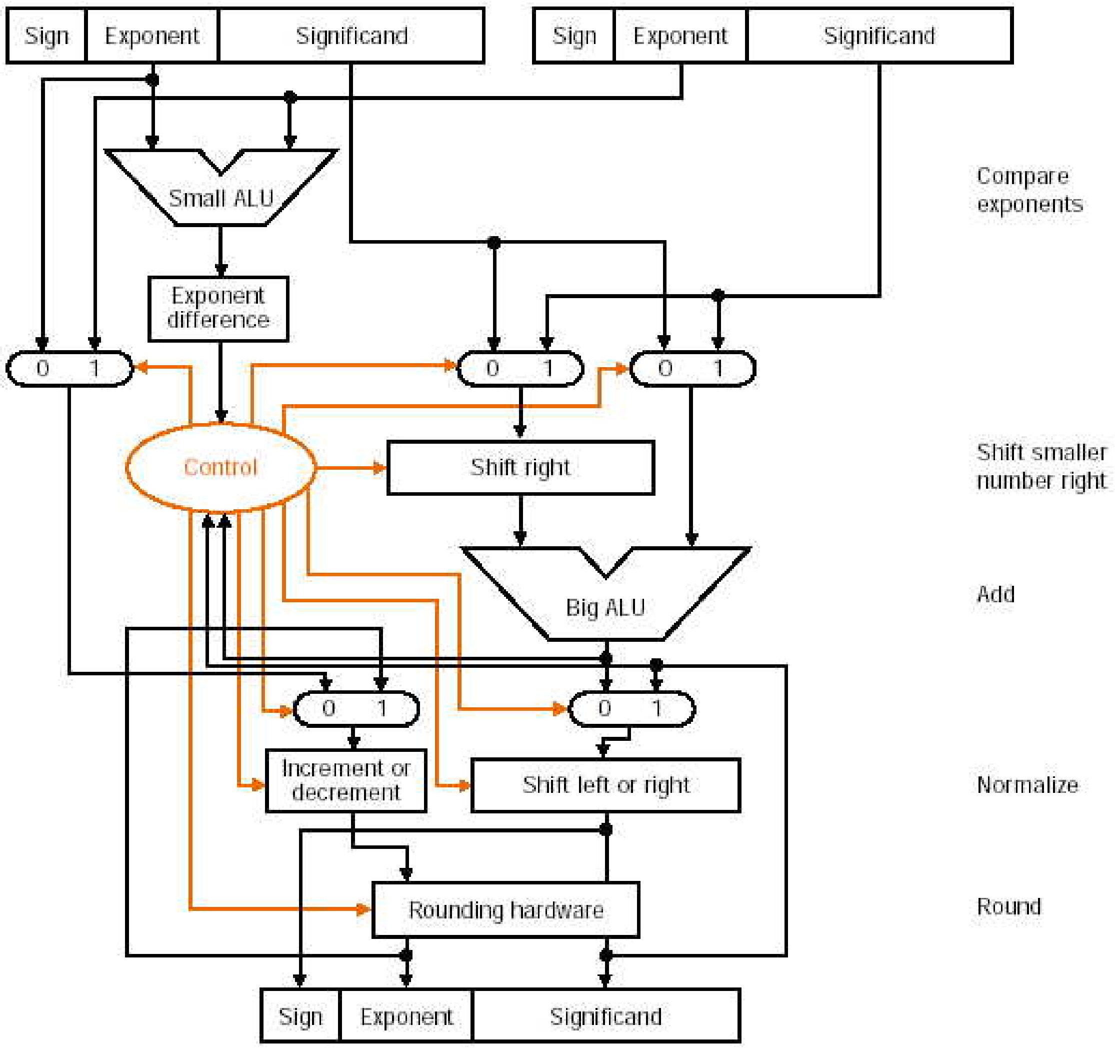

Figure 5: Floating-point adder block diagram

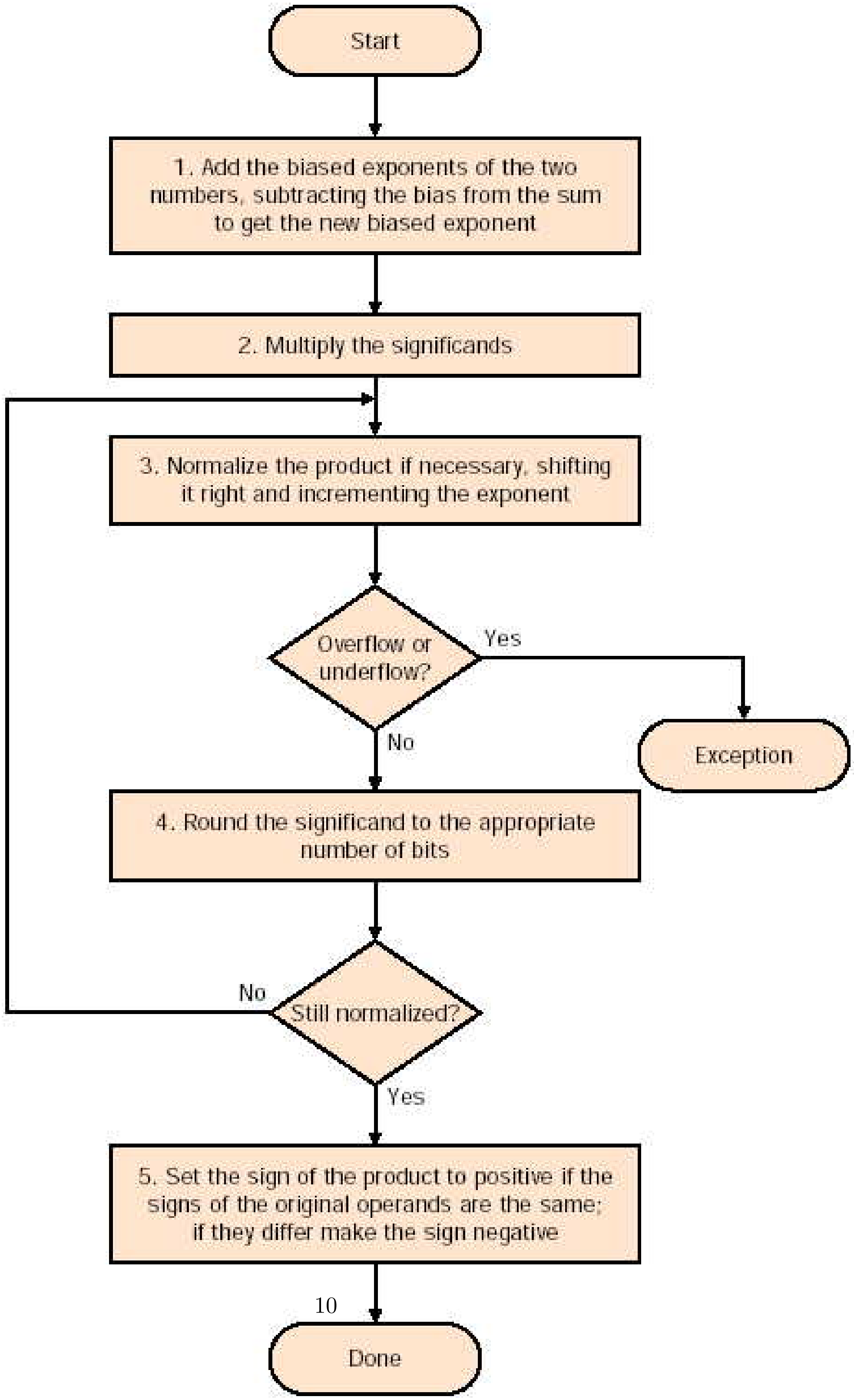
Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Figure 6: Floating-point multiplier flowchart