**Arithmetic-Logical-Floating-Point Unit**

**performing subtraction and division**

Student: Ana-Sofia Cândea

Structure of Computer Systems Project

Technical University of Cluj-Napoca

Contents

1. **Introduction**
   1. Context
   2. Objectives
2. **Bibliographical Research**
   1. What is an ALU?
   2. Floating-Point Subtraction
   3. Floating-Point Division
3. **Analysis**
   1. User Diagram
   2. Floating-Point Subtraction Analysis
   3. Floating-Point Division Analysis
4. **Design**
   1. General Overview
   2. Floating-Point Subtraction Design
   3. Floating-Point Division Design
5. **Implementation**
   1. Subtraction
   2. Division
6. **Testing & Validation**
   1. Subtraction
   2. Division
7. **Conclusions**
8. **Bibliography**

**Introduction**

* 1. **Context**

The aim of this project is to design an Arithmetic-Logical-Floating-Point Unit on 32 bits which implements the subtraction and division operations. In modern computer architecture, Floating-Point Units are a specialized part of an Arithmetic Logic Unit (ALUs), handling operations on floating-point numbers according to standards like IEEE 754.

Division and subtraction operations are often more complex and require higher precision compared to simpler operations like addition or multiplication. Therefore, an optimized design is crucial for improving overall computational performance and reducing latency, particularly in systems where floating-point operations are heavily utilized.

* 1. **Objectives**

The unit will be designed in VHDL and included in a Xilinx Vivado Project. The Floating-Point Unit (FPU) will be integrated into a larger processing unit. The final design will incorporate principles of pipelining and hazard avoidance to ensure efficiency and accuracy.

**Bibliographical Research**

**2.1 What is an ALU?**

An Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU) responsible for performing arithmetic and logical operations. These operations include basic arithmetic (addition, subtraction, multiplication, division) and logic functions (AND, OR, XOR, NOT). The ALU receives inputs in the form of binary data from the processor's registers, processes them based on the control signals, and outputs the result. It plays a crucial role in executing instructions and is essential for the overall functionality of digital computing systems.

**2.2 Floating Point Arithmetic**

Floating-point numbers allow a wide range of values by representing numbers in a normalized form, with a mantissa and an exponent. The IEEE 754 standard specifies a binary representation for floating-point numbers to support a wide range of values, ensuring consistent precision and predictability across different systems. A 32-bit IEEE 754 floating-point number (also called "single-precision") is divided into three parts:

* **Sign Bit**: 1 bit, indicating the sign of the number (0 for positive, 1 for negative).
* **Exponent**: 8 bits, which represents the exponent, biased by a constant value of 127. This bias allows representation of both positive and negative exponents.
* **Mantissa (Significand)**: 23 bits, representing the fractional part of the number. The IEEE 754 format assumes a leading "1" (implicit bit) for normalized numbers, so only the fractional bits are stored.

**2.3 Floating Point Subtraction**

Subtraction poses challenges primarily due to the need to align the exponents of the operands before performing the operation. After that, the adjusted mantissas are subtracted and a normalization of the result might be necessary. If the exponents differ significantly, the mantissas must be shifted, which may lead to a loss of precision. Techniques to minimize this precision loss, such as guard bits and rounding strategies, are critical in ensuring accuracy​.

**2.4 Floating Point Division**

Division is among the most time-consuming operations in floating-point arithmetic, often requiring iterative methods such as Newton-Raphson or Goldschmidt’s algorithm to approximate the result. The algorithm iterates to refine this approximation until a desired precision is achieved. Each iteration reduces the error by a factor, often halving it. Once a suitable precision level is reached, the mantissa may need to be normalized, and rounding is applied as per IEEE 754. These methods can achieve high precision but may require a significant number of iterations. Recent research has focused on optimizing these algorithms to reduce the number of cycles needed for convergence​.

**Analysis**

**3.1 User Diagram**

A diagram of a person with arrows

Description automatically generated

Use-case: Select operation and introduce the values

* Primary Actor: User
* Main Success Scenario:
* The user selects the operation
* The user introduces the 2 numbers
* The operation is performed, and the result is displayed

**3.2 Floating-Point Subtraction Analysis**

Floating-point subtraction requires aligning the operands’ exponents before performing the operation on the mantissas. Here’s a step-by-step breakdown:

* + 1. **Aligning Exponents**:
* Compare the exponents of both operands. If the exponents differ, the operand with the smaller exponent has its mantissa right shifted until its exponent matches the larger one.
* Each shift decreases precision slightly, so this is where precision loss can occur, particularly if there is a large difference between the exponents
  + 1. **Performing Subtraction on Aligned Mantissas**:
* After alignment, the mantissas are subtracted. The result’s sign depends on the original signs and the relative magnitudes of the operands.
* If the operands have different signs, the mantissas are added (as it’s equivalent to adding the absolute values).
  + 1. **Normalization**:

After subtraction, the resulting mantissa may need to be normalized. Normalization ensures that the result mantissa has a leading 1 (in IEEE 754, this leading bit is implicit).

* **Shift Mantissa**: If the result mantissa does not begin with a 1, shift it left until it does, adjusting the exponent accordingly.
* **Adjust Exponent**: Each left shift of the mantissa decreases the exponent by one.
  + 1. **Rounding and Guard Bits**:
* Guard bits are extra bits added during shifts to maintain precision. If the mantissa has excess bits after alignment or subtraction, guard bits ensure these bits aren’t lost immediately.
* After calculating the result, the mantissa is rounded to fit the 23-bit limit of the IEEE 754 single-precision format. Rounding modes like round-to-nearest help achieve the most accurate result possible.

**Example:**

Suppose we have two numbers:

1. A=5.5 and B=1.25, represented in IEEE 754 32-bit binary as:
   * A=
   * B=

**Step-by-Step Calculation in Binary**

1. **Align Exponents:**
   * A= (which is in scientific notation).
   * B= (which is in scientific notation).
   * To align B with A's exponent, we shift B's binary point two places to the right:
   * B becomes **0.0101** (aligned with exponent 2), which in binary scientific notation is
2. **Subtract Mantissas:**
   * We now have:
     + A=1.011 (which is 101.1 in binary, equivalent to 5.5 in decimal)
     + B=0.0101 (aligned version of 1.25, now represented as in binary)
   * Perform the binary subtraction: 1.011−0.0101=1.0001 So the result is **1.0001** in binary.
3. **Intermediate Result:**
   * After subtraction, the intermediate result is (representing 4.25 in decimal).
4. **Normalization:**
   * The result,, is already in normalized form, so no further adjustments are needed.
5. **Convert to IEEE 754 Format:**
   * **Sign Bit**: 0 (since the result is positive)
   * **Exponent**: (adding the IEEE 754 bias of 127)
     + 129 in binary is **10000001**.
   * **Mantissa**: Encode the fractional part of **1.0001** as **00010000000000000000000** (filling remaining bits with zeros up to 23 bits).
6. **Final IEEE 754 Representation:**
   * Combining everything, we get the IEEE 754 binary format for 4.25 as: **0 10000001 000100000000000000000000**

**3.3 Floating Point Division Analysis**

Floating-point division in IEEE 754 format is more complex than subtraction due to the need for iterative calculation of the quotient, careful handling of exponents, and potential normalization of the result. This section provides a step-by-step breakdown of the division process, followed by an example to demonstrate each step.

This process involves the following steps:

* + 1. **Sign Calculation**
* The result sign is positive if both operands have the same sign and negative if the operands have different signs.
* This can be determined with a simple XOR operation on the sign bits.
  + 1. **Exponent Calculation**
* **Calculate the Difference**: Subtract the exponent of B from the exponent of A.
* **Bias Adjustment**: Add the IEEE 754 bias (127 for single precision) to the result exponent.

This adjustment ensures that the result exponent remains within the valid IEEE 754 range.

* + 1. **Mantissa Division**
* **Normalize the Mantissas**: Both mantissas are normalized to start with an implicit leading 1.
* **Perform Iterative Division**: Use a division algorithm, such as non-restoring division, to divide the mantissa of A by the mantissa of B. Each iteration produces a bit of the quotient, accumulating until the full 23-bit mantissa is obtained.
  + 1. **Normalization**
* If the result mantissa is not normalized (i.e., doesn’t have a leading 1), shift it left until normalization is achieved**.**
* Each left shift decreases the exponent by 1.
  + 1. **Rounding**
* If the normalized mantissa exceeds 23 bits, rounding is applied to fit within the IEEE 754 format. Round-to-nearest is commonly used, where the extra bits are inspected to determine if the mantissa should be rounded up.

**Example:**

Let's go through each step with an example where A=10.0 and B=2.0.

1. **Convert to IEEE 754 Format**:
   * A=10.0:
     + **Sign**: 0 (positive)
     + **Exponent**: 130 (binary: 10000010) represents (130 - 127 = 3)
     + **Mantissa**: 1.01000000000000000000000 (binary representation of 1.25 in decimal)
   * B=2.0:
     + **Sign**: 0 (positive)
     + **Exponent**: 128 (binary: 10000000) represents (128 - 127 = 1)
     + **Mantissa**: 1.00000000000000000000000
2. **Step-by-Step Calculation**:

**Step 1: Calculate Sign**

* + **Sign of Result** =
  + The result is positive.

**Step 2: Calculate Exponent**

* + **Exponent Difference** =
  + **Bias Adjustment**: Add 127 to the difference:

**Step 3: Divide the Mantissas**

* + **Mantissa of A**: 1.01000000000000000000000 (binary)
  + **Mantissa of B**: 1.00000000000000000000000 (binary)
  + Perform binary division on the mantissas:
  + **Result Mantissa** = 1.01000000000000000000000

**Step 4: Normalize the Result**

* + The result mantissa 1.01000000000000000000000 is already normalized, so no adjustment is needed.

**Step 5: Rounding**

* + The result mantissa is within the 23-bit limit, so no rounding is required.

1. **Final IEEE 754 Representation**:
   * **Sign**: 0
   * **Exponent**: 129 (after bias adjustment)
   * **Mantissa**: 010000000000000000000000 (with the leading 1 implicit)

The final IEEE 754 binary representation of is:

**0 10000001 01000000000000000000000**

**Design**

**4.1 General Overview**

A white square with black text

Description automatically generated

Figure

This Floating-Point Arithmetic Logic Unit (ALU) is designed to perform IEEE 754 single-precision operations, specifically focusing on subtraction and division. It receives two 32-bit floating-point inputs, A and B, and outputs a 32-bit result that adheres to IEEE 754 format. The ALU manages key floating-point operations, including exponent alignment, mantissa comparison, iterative division, normalization, and rounding, ensuring precise and efficient calculations over a wide range of values.

**4.2 Floating Point Subtraction Design**

The subtraction unit design can be divided into multiple modules. First, we need to handle the exponent alignment, then the mantissa subtraction and then the rounding and normalization. In order to perform each operation, we can define the following functional blocks:

A diagram of a blue rectangle with green lines

Description automatically generated

Figure

* + 1. **Exponent Alignment**

The first step of the subtraction operation between two floating point numbers is to align the exponents. This component takes two 8-bit exponent inputs (expA and expB) and two 24-bit mantissa inputs (mantA and mantB). The component adjusts the mantissas so that they can be added or subtracted accurately, by shifting the mantissa of the number with the smaller exponent to the right.

* + 1. **Mantissa Comparator**

This component is designed to compare the mantissas of two 24-bit floating-point numbers. It outputs signals indicating the relationship between the two mantissas, specifically whether the first mantissa is greater than, less than, or equal to the second. The comparison result may be used to determine the order of subtraction (which number should be subtracted from which).

* + 1. **Mantissa Subtractor**

In order to ensure a proper functionality of the subtraction operation, we need to define multiple blocks for this operation:

* **Adder:**

This is part of the mantissa subtraction module. The adder is used to perform the addition between the first number and the two’s complement of the second one. This way we can perform the subtraction operation. It outputs the sum and the carry-out, indicating overflow (if exists).

* **Two’s Complement Converter:**

This is part of the mantissa subtraction module. This component converts a binary number into its two's complement form. It takes the binary representation of a floating-point number and converts it into its negative equivalent by inverting the bits and adding one.

* + 1. **Normalizer**

The Normalizer is a VHDL component responsible for normalizing the mantissa of a floating-point number based on its value. It ensures that the mantissa is adjusted to fit the standard normalized form as defined by the IEEE 754 floating-point representation.

All the blocks are then linked together in order to perform the subtraction of two floating point numbers. The main file contains all the components and it consist of he core functionality of the ALU for performing floating-point subtraction. It takes wo floating-point inputs (in IEEE 754 format) and calculates their difference, handling the alignment of the exponents and performing the subtraction on the mantissas, ensuring that the returned result is in the appropriate floating-point format, including normalization if necessary.

* 1. **Floating Point Division Design**

A diagram of a computer network

Description automatically generated

Figure

The design can be broken down into multiple functional blocks to manage exponent calculation, iterative mantissa division, normalization, rounding, and special case handling. Each of these blocks ensures that the division result adheres to the IEEE 754 single-precision format.

* + 1. **Sign Calculator**

This component determines the sign of the result by comparing the signs of the two operands. The IEEE 754 standard represents positive numbers with a sign bit of 0 and negative numbers with a sign bit of 1. The result sign is calculated by XOR-ing the sign bits of the dividend (A) and divisor (B). If both operands have the same sign, the result is positive; if they have different signs, the result is negative.

* + 1. **Exponent Calculator**

This component handles the exponent calculation independently, preparing the result for the next steps in the division process. It computes the result exponent by subtracting the exponent of the divisor from that of the dividend and adjusting it for the IEEE 754 bias. Division reduces the magnitude, so the divisor’s exponent is subtracted from the dividend’s exponent. The IEEE 754 format uses a bias of 127, which is added back after exponent subtraction to maintain the correct scaling.

* + 1. **Mantissa Divider**

The Mantissa Divider provides the main division logic, isolating the iterative division process from other parts of the design. The component performs bit-by-bit division using the non-restoring division algorithm:

* + 1. **Normalizer**

It guarantees that the mantissa meets the IEEE 754 requirements before rounding, preserving precision and compliance with floating-point standards. If the most significant bit (MSB) of the result mantissa is 0, the mantissa is unnormalized. The component left-shifts the mantissa until a leading 1 is obtained, adjusting the exponent downward with each shift to maintain the correct value.

**Implementation**

**5.1 Subtraction**

The implementation stage for the subtraction part starts with adjusting and testing the separate components of the Floating Point Subtractor. Each component is designed to implement a specific requirement of the subtractor.

**5.1.1 Fetching Data from Register**

In the floating-point subtraction process, the register file component is responsible for holding and providing data for the operands. The register file allows the system to access different values stored in specific addresses and is particularly useful for retaining intermediate data or constants required in arithmetic operations.

This design enables the floating-point subtractor to easily access and retrieve the required operand from the register file for the subtraction operation.

A diagram of a computer network

Description automatically generated

**5.1.2 Preparing for subtraction**

After the values are fetched, the preparation part is next. Before computing the subtraction, the numbers are separated into sign, exponent and mantissa and each one is taken separately.

A diagram of a mantisal separator

Description automatically generated

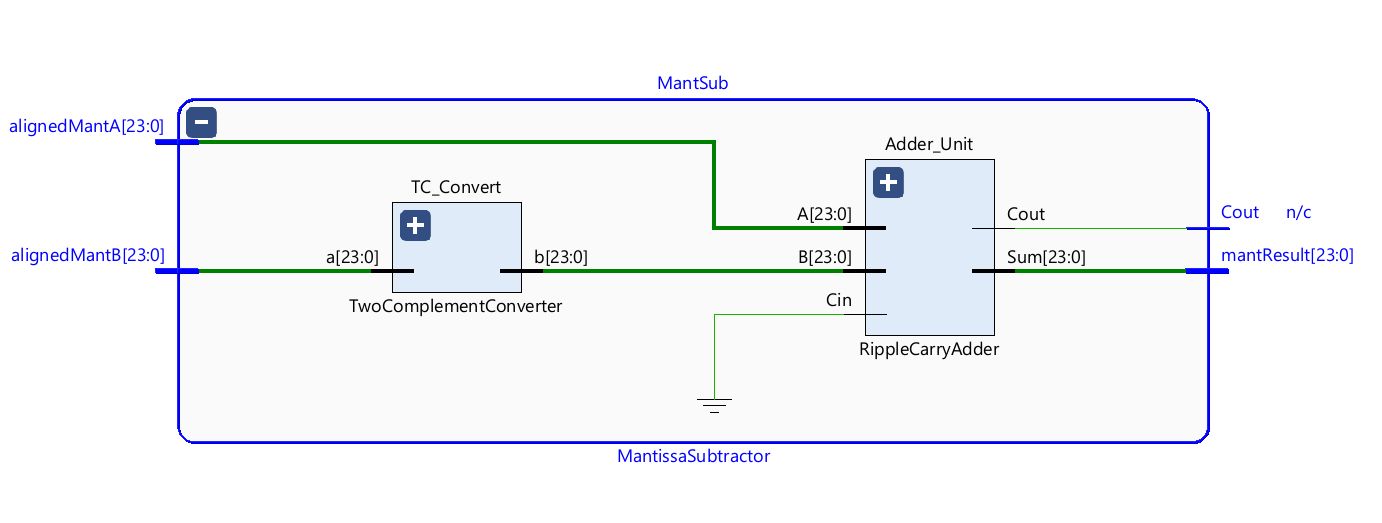
**5.1.3 Aligner**

Based on the exponents, the mantissas are aligned, preparing them for comparison and subtraction. The exponent with a larger value is stored as the result exponent.

After these steps, the subtraction part of the mantissa can begin.

**5.1.4 Mantissa Subtraction**

This part of the implementation handles the subtraction of mantissas. In order to achieve the correct result and to maximize efficiency, a Ripple Carry Adder on 23 bits is used. This is composed of a full adder cascaded 24 times. A Two’s Complement Converter handles the conversion of the operand which is to be subtracted.



**5.1.5 Normalizer**

The last part is to normalize the result in order to ensure that the mantissa is adjusted to fit the standard normalized form as defined by the IEEE 754 floating-point representation.

**5.2 Division**

The implementation stage for the division part starts with adjusting and testing the separate components of the Floating Point Divider.

**5.2.1 Preparing for Division**

In this phase, the input values are fetched from the register file, and similar to subtraction, the values are separated into:

* **Sign**: The sign bits are isolated from both operands.
* **Exponent**: The exponents are extracted and prepared for calculation.
* **Mantissa**: The implicit leading '1' is restored to the mantissa for further division operations.

The next step involves calculating the **result sign** and the **result exponent**.

* **Sign Calculation**: Since our design is working only with positive numbers, the result will return a positive number as well. There is no need to handle the sign separately;
* **Exponent Calculation**: The exponents are adjusted by subtracting the exponent of the divisor (B) from the exponent of the dividend (A). The IEEE 754 bias (127) is added to normalize the exponent:

At this stage, both mantissas are ready for division.

A computer screen shot of a diagram

Description automatically generated

**5.2.2 Mantissa Division**

The core logic for floating-point division is implemented using a non-restoring division algorithm. This iterative process produces the result mantissa one bit at a time.

* Initialization:
* The mantissa of the dividend (A) is set as the initial remainder.
* The divisor's mantissa (B) is used as the divisor in the iterative process.
* The result quotient (mantissa) is initialized to zero and a scaling is performed so that it has a width of 48 bits.
* Iterative Division: The non-restoring division works as follows:
* Shift the Remainder and Quotient left by one bit.
* Subtract the Divisor: Subtract the divisor's mantissa from the remainder.
* If the result is positive, the current quotient bit is set to 1.
* If the result is negative, the remainder is restored (add back the divisor), and the current quotient bit is set to 0.
* This process continues for 48 iterations to generate a 23-bit mantissa with an additional guard bit for rounding.
* Final Mantissa: At the end of the iterations, the mantissa represents the division result. However, it may not be normalized.

**A diagram of division and division

Description automatically generated**

**5.2.3 Normalization**

The Normalizer ensures that the mantissa adheres to the IEEE 754 format by adjusting it to start with an implicit leading '1'.

* If the mantissa does not have a leading '1', it is shifted left until it does.
* Each left shift decreases the result exponent by 1 to preserve the value.
* The final normalized mantissa is truncated to 23 bits.

**5.2.4 Final Assembling**

After the sign, exponent, and mantissa have been calculated and normalized, they are assembled into the final IEEE 754 32-bit format:

* Sign Bit: The calculated result sign.
* Exponent: The adjusted result exponent.
* Mantissa: The 23-bit normalized mantissa (excluding the implicit leading '1').

The final result is then shown.

**Testing and Validation**

In order to test the functionality of the algorithm, different floating point values were used as addresses stored in the registers. This resulted in the following test cases:

1. **Subtraction:**
   1. Test case 1:

A=236, 529 ( hex value: 436c87d )

B= 0,06814 (hex value: 3d8bcfc )

Expected Result = 236,461 (hex value: 436c75fc )

A screenshot of a computer

Description automatically generated

* 1. Test case 2:

A= 0,890238 (hex value: 3f63e6a5)

B=0,06814 (hex value: 3d8b8cfc )

Expected Result = 0.822098 (hex value : 3f527206)

A screenshot of a computer

Description automatically generated

* 1. Test case 3:

A=236,529 (hex value: 436c876d)

B=125,003 (hex value: 42fa0189)

Expected Result = 111.526(hex value: 42df0d52)

A black and green line

Description automatically generated

* 1. Test case 4:

A=20,25 (hex value: 41a20000)

B=5,0 (hex value:40a00000)

Expected Result = 15,25 (hex value: 41740000)

A screenshot of a computer

Description automatically generated

* 1. Test case 5:

A=20,25 (hex value: 41a20000)

B=0,890238 (hex value: 3f63e6a5)

Expected Result =19.3598 (hex value: 419ae0cb)

A screenshot of a computer

Description automatically generated

* 1. Test case 6:

A= -38.927853 (hex value: c21bb61f)

B= -12597.5443 (hex value c644d62d)

Expected Result= 12558.6164 ( hex value: 46443a77)

A screenshot of a computer

Description automatically generated

1. **Division**
   1. Test case 1:

A=20,25 (hex value: 41a20000)

B=0,890238 (hex value: 3f63e6a5)

Expected result= 22.7467 (hex value: 41b5f94a)

A green lines on a black background

Description automatically generated

* 1. Test case 2:

A=397,388 (hex value: 43c6b1aa)

B=0,890238 ( hex value: 3f63e6a5)

Expected result= 446,385 (hex value: 43df3124)

A screenshot of a video game

Description automatically generated

* 1. Test case 3:

A=0,021 (hex value: 3cac0831)

B=0,034 (hex value: 3d0b4396)

Expected result= 0,617647 (hex value: 3f1e1e1d)

A screenshot of a computer

Description automatically generated

* 1. Test case 4:

A=20,25 (hex value: 41a20000)

B=20,25 (hex value: 41a20000)

Expected result= 1 (hex value: 3f800000)

A screenshot of a video game

Description automatically generated

* 1. Test case 5 (exception: error of division by 0):

A=20,25 (hex value: 41a20000)

B=0

Expected result= error

A screenshot of a video game

Description automatically generated

* 1. Test case 6

A= -38.927853 (hex value: c21bb61f)

B= -12597.5443 (hex value c644d62d)

Expected result= 0.0030901143( hex value: 3b4a8384)

A black and green screen

Description automatically generated with medium confidence

**Conclusions**

The design and implementation of the 32-bit Arithmetic-Logical-Floating-Point Unit (ALU), focused on floating-point subtraction and division, provided significant insights into the complexities of precision arithmetic in digital systems. By adhering to the IEEE 754 standard, the project demonstrated how floating-point operations can be efficiently managed while ensuring accuracy and minimizing precision loss.

In conclusion, this project successfully achieved its objectives of designing a high-precision floating-point ALU capable of performing subtraction and division operations. The modular structure and adherence to industry standards make the design well-suited for integration into modern computing architectures, where efficient and precise arithmetic is essential. Future work could explore expanding the ALU to handle additional operations, such as square root or trigonometric functions, as well as optimizing the hardware further for specific application domains.

**Bibliography**

1. G. Sebestyen, “SCS\_course\_3”, Technical University of Cluj Napoca, (2024)
2. Wikipedia, “IEEE 754”, <https://en.wikipedia.org/wiki/IEEE_754>
3. M.Bhavani, G.Sirisha, K.Siva Kumari, B.Lalitha Rani, A.Charishma, “Design of Single Precision Floating Point Arithmetic Logic Unit”, International Journal of Engineering Research and Applications (2011)
4. Dr Ranjani Parthasarathi, “Floating Point Arithmetic Unit”, Computer Architecture (2010)
5. Geeks for geeks, “IEEE-Standard-754-floating-point-numbers” <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers>