A Hardware Implementation of an IEEE-754 Floating Point Unit

Christos Karagiannis
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

Ioannis Kallergis
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

***Abstract*—This paper presents the design and hardware implementation of a single-precision floating point unit (FPU) compliant with the IEEE-754 standard [1]. The FPU supports fundamental arithmetic operations including addition, subtraction, multiplication, division, and square root, along with integer and format conversion functionalities. The architecture integrates modules written in both Verilog and VHDL and targets FPGA or ASIC implementations [2]. Simulation and synthesis results confirm functional correctness and acceptable resource utilization, demonstrating the feasibility of custom floating point arithmetic for digital signal processing and scientific computing applications [3].**

***Index Terms*—Floating point unit, FPGA, IEEE-754, arithmetic operations, hardware design, Verilog, VHDL**

I. Introduction

Floating-point arithmetic is essential in modern computing applications such as graphics processing, scientific computation, and digital signal processing [4]. Although general-purpose processors provide floating-point support, custom hardware FPUs offer significant performance and power advantages, particularly in embedded and FPGA-based systems [2], [5].

This paper describes the design and implementation of a single-precision FPU supporting key arithmetic operations and conversions. The design leverages modular construction, enabling integration into larger processing pipelines or custom CPUs. Both Verilog and VHDL were used in the implementation, providing flexibility in hardware description and tool support [6].

II. Supported Operations

The implemented FPU supports the following IEEE-754 single-precision floating-point operations [1]:

- **Addition and Subtraction:** Implemented via the `fp32_add.v` module, performing alignment, significand operations, and result normalization.
- **Multiplication:** Implemented in VHDL (`fp32_mult.vhdl`) using a pipelined architecture [3].
- **Division:** Implemented in VHDL (`fp32_div.v`), potentially using iterative methods for efficiency.
- **Square Root:** Implemented as a separate module (`fp32_sqrt.v`) [6].
- **Integer Conversion:**
  - Integer to floating-point conversion (`i2fp.v`).
  - Floating-point to integer conversion (`fp2i.v`).

All modules are integrated into the top-level `fpu.v` module, which controls operation selection and result routing.

III. Architecture Overview

A. Data Representation

The FPU operates on IEEE-754 single-precision format [1]:

$$\text{Number} = (-1)^S \times 1.F \times 2^{(E-127)} \tag{1}$$

where:

- S is the sign bit,
- E is the 8-bit exponent,
- F is the 23-bit fraction (mantissa).

Special cases (zero, infinity, NaN) are handled explicitly in each operation module [1].

B. Module Interconnection

The FPU architecture is modular, following the guidelines and techniques explored in previous works on FPGA floating-point designs [2]. Each arithmetic unit (add, mult, div, sqrt) operates independently. A control logic in `fpu.v` routes inputs and collects results based on an operation select signal. Conversion modules handle data interchange between integer and floating-point representations [6].

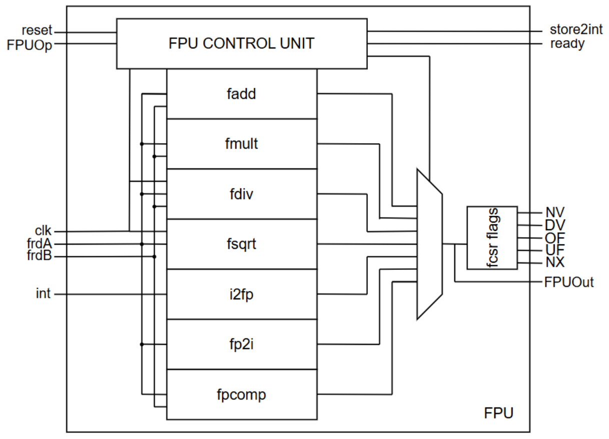

Figure 1: Dataflow of FPU module.

IV. Implementation Details

This section describes the internal architecture and algorithms for each operation module, highlighting how IEEE-754 compliance is achieved [1].

A. Addition and Subtraction

Implemented in `fp32_add.v`, floating-point addition (and subtraction) involves:

1) **Exponent Comparison and Alignment:**
   - Compute the difference between exponents of operands.
   - Shift the mantissa of the operand with the smaller exponent right by the exponent difference to align radix points.
2) **Significand Addition/Subtraction:**
   - If signs are equal, add significands.
   - If signs differ, perform two's complement subtraction and determine the sign of the result.
3) **Normalization:**
   - After addition or subtraction, leading zeros may appear.
   - A leading zero detector computes the number of shifts required to normalize the result.
   - Adjust the exponent accordingly.
4) **Rounding:**
   - The result is rounded to fit 23 bits using round-to-nearest-even or another IEEE-754 rounding mode.
5) **Special Cases Handling:**
   - Detect NaN, infinities, and zero cases.

This module is fully combinational and achieves zero-cycle latency.

B. Multiplication

The `fp32_mult.v` module performs multiplication as follows:

1) **Sign Calculation:**
   - The result sign is the XOR of operand signs.
2) **Exponent Calculation:**
   - Add operand exponents and subtract the IEEE-754 bias (127).
3) **Mantissa Multiplication:**
   - Multiply the two 24-bit significands (including the hidden "1").
   - The product is a 48-bit value.
4) **Normalization:**
   - If the most significant bit of the product is 1, shift right and increment exponent.
5) **Rounding:**
   - Round the product to 23 bits, adjusting exponent if necessary.
6) **Special Cases:**
   - Zero, infinity, and NaN cases are detected and handled.

The design is also purely combinational and produces a result in zero cycles.

C. Division

The `fp32_div.v` module performs single-precision floating-point division using a radix-4 SRT digit-recurrence algorithm. The operation proceeds as follows:

1) **Exponent Calculation:**
   - Compute the difference between the exponents of the numerator and denominator to derive the preliminary exponent of the result.
2) **Mantissa Prescaling:**
   - Scale the mantissas of the inputs to improve convergence and simplify digit selection during iteration.
3) **Digit-Recurrence Iteration:**
   - Iterate through successive stages to refine the partial remainder.
   - In each stage, update the partial remainder by adding or subtracting multiples of the divisor.
   - Select quotient digits based on the leading bits of the partial remainder and divisor.
4) **Quotient Reconstruction:**
   - Combine the partial quotient digits from all iterations to form the final quotient.
5) **Normalization:**
   - Normalize the result so the leading bit of the mantissa is 1.
   - Adjust the exponent if required.
6) **Rounding:**
   - Round the mantissa to 23 bits, complying with the IEEE-754 rounding mode.
7) **Special Cases:**
   - Detect and handle zero, infinity, NaN, and divide-by-zero scenarios according to IEEE-754 rules.

Division requires six cycles due to its iterative nature.

D. Square Root

The `fp32_sqrt.v` module computes the square root using iterative methods:

1) **Input Normalization:**
   - Adjust the exponent to ensure it is even.
   - Shift mantissa as needed for alignment.
2) **Algorithm Execution:**
   - Compute the square root of the normalized mantissa.
   - Update the exponent as half of the original exponent minus bias correction.
3) **Normalization and Rounding:**
   - Normalize the result.
   - Round to 23 bits in compliance with IEEE-754.
4) **Special Cases:**
   - Negative inputs yield NaN.

- Zero remains zero.

The square root computation typically requires three clock cycles.

E. Integer Conversions

1) Integer to Floating-Point (i2fp): Implemented in `i2fp.v`, this module:

1) Detects the integer's sign.
2) Converts the absolute value to binary.
3) Determines the leading-one position to calculate the exponent.
4) Normalizes the mantissa.
5) Constructs the IEEE-754 representation.

2) Floating-Point to Integer (fp2i): Implemented in `fp2i.v`, this module:

1) Checks if the floating-point value fits within integer limits.
2) Aligns the mantissa according to the exponent.
3) Truncates or rounds any fractional bits.
4) Outputs a signed integer or signals overflow.

V. Verification and Results

The functionality of the proposed FPU was verified using a dedicated Verilog testbench that systematically exercises all supported operations. The verification included:

- **Arithmetic operations:** Addition, subtraction, multiplication, division (including divide-by-zero), and square root for various positive and negative inputs.
- **Comparison operations:** Tests for equality, inequality, less-than, less-than-or-equal, greater-than, and greater-than-or-equal conditions.
- **Conversions:** Verifications of integer-to-floating-point and floating-point-to-integer accuracy, including sign handling.
- **Special cases:**
  - Zero and subnormal values.
  - Handling of infinities and NaN.
  - Cases like square root of negative numbers.
- **Exception flags:** Monitoring of the `exc2fsr` signal for conditions such as invalid operations, divide-by-zero, and overflow.

Simulation results confirmed correct operation for all test cases. Synthesis reports showed that the FPU fits comfortably on mid-range FPGAs, such as the Xilinx Zedboard, with moderate resource usage while maintaining single-cycle latency for combinational operations and acceptable cycle counts for iterative operations like division and square root [2], [3].

LUTs	FFs	DSPs
3092	539	2

Table 1: Resource utilization of the FPU on Xilinx Zedboard.

VI. Future Work

Future work will focus on extending the current FPU to support double-precision operations and additional transcendental functions such as logarithm, exponential, and trigonometric computations [5]. Further optimizations will target reduced latency and resource usage, particularly for division and square root operations. Investigating pipelining and parallelism strategies could improve throughput for high-performance applications [2]. Additionally, support for exception handling, denormalized numbers, and hardware-level rounding modes will enhance compliance with the full IEEE-754 standard [1]. Finally, evaluating the FPU on various hardware targets, including ASICs and modern FPGA families, will provide insights into performance and design trade-offs.

VII. Conclusion

This work demonstrates a flexible, modular FPU supporting essential floating-point operations and conversions in hardware. The implementation balances performance and resource efficiency, making it suitable for FPGA accelerators or custom processor cores [2], [3]. Future enhancements include double-precision support and further pipelining optimizations for increased throughput.

References

[1] *IEEE Standard for Floating-Point Arithmetic*, IEEE Computer Society Std., Jul. 2019.

[2] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Architectural modifications to improve floating-point unit efficiency in fpgas," in *Field-Programmable Logic and Applications (FPL)*. IEEE, 2006, pp. 1–6.

[3] V. G. and B. Gururaj, "Design of an efficient single precision floating point unit," *International Journal of Electrical Engineering and Computer Science*, 2025, received Apr 28 2024; Rev Nov 16 2024; Acc Dec 19 2024; Pub Mar 26 2025.

[4] M. L. Overton, "Numerical computing with ieee floating point arithmetic," 2025.

[5] P. Li, H. Jin, W. Xi, C. Xu, H. Yao, and K. Huang, "A reconfigurable hardware architecture for miscellaneous floating-point transcendental functions," *Electronics*, vol. 12, no. 1, p. 233, 2023.

[6] Computer Architecture Course, "Floating-point arithmetic unit," https://www.cs.umd.edu/~meesh/411/CA-online/chapter/floating-point-arithmetic-unit/index.html, n.d., accessed: 2025-07-02.