# VHDL Model of a Half-Precision Floating-Point Arithmetic Unit with Remainder and Rounding

Glenyvie A. Frondozo, Klarisse C. Odiame, Daryl Ann T. Respeto, and Joel R. Noche*
Department of Electronics and Computer Engineering
Ateneo de Naga University
Naga City, 4400 Philippines
*jnoche@gbox.adnu.edu.ph*

Date received: November 13, 2025
Revision accepted: April 13, 2026

## Abstract

*Deep learning inference can be accelerated by using field-programmable gate arrays (FPGAs) and reduced-precision floating-point types, but existing half-precision arithmetic unit designs often do not fully comply with the IEEE 754-2008 standard, particularly in rounding and remainder operations. This paper describes a model of a synchronous, variable-latency, half-precision, floating-point arithmetic unit written in VHDL and synthesizable on an FPGA. The model, based on previous work, performs addition, multiplication, division, and remainder, and rounds the result using four of the five IEEE 754-2008 rounding modes. Its rounding algorithm is discussed in detail. The model was tested using 246 test vectors representing four operations, five floating-point data, five exceptions, and four rounding modes, simulated using Lattice Diamond 3.10 and Active-HDL 10.3. The model yielded correctly-rounded results to 244 of 246 test vectors. The two vectors that yielded inexact results involved a very large number multiplied by a very small number, attributable to insufficient intermediate storage width. Completion times with rounding ranged from 1 to 39 clock cycles for addition, 1 to 62 for multiplication, 1 to 102 for division, and 1 to 60 for remainder. The model has since been extended to single- and double-precision variants and verified on an FPGA, confirming its scalability across floating-point formats and its potential for use in deep learning inference applications.*

***Keywords**:* adders, coprocessors, floating-point arithmetic, multiplying circuits, VHDL

## 1. Introduction

Floating-point numbers are currently the most common way to approximate real numbers in computers (Boldo *et al*., 2023). The IEEE (1985) floating-point arithmetic standard has greatly facilitated the design and portability of

numerical software. Applications such as digital neural network training need a huge number of calculations, making smaller floating-point formats (such as 16-bit) increasingly popular (Boldo *et al*., 2023).

Noche (2003) designed and tested (using Cadence software) an arithmetic unit that handles addition, multiplication, division, and remainder following the IEEE 754-1985 Standard for Binary Floating-Point Arithmetic (IEEE, 1985). It is a transistor-level design, handles single-precision numbers, is asynchronous (doesn't use a global clock signal), and has no rounding unit.

Noche and Araneta (2007) possibly authored the only published work on floating-point addition design utilizing asynchronous circuits at the time, as recognized by Sheikh and Manohar (2010). This work was recognized as the first asynchronous implementation of a single-precision floating-point adder, encompassing other arithmetic operations, whereas previous implementations of floating-point units primarily concentrated on multiplication or division functionalities (Srivastava *et al*., 2020).

Frondozo *et al*. (2020) modified Noche's design to be described with the hardware description language VHDL, support half-precision numbers, operate synchronously, and include a rounding unit. Table 1 summarizes the main differences between the two designs.

Table 1. Comparison of the designs of Noche (2003) and of Frondozo *et al*. (2020)

| | Noche (2003) | Frondozo *et al*. (2020) |
|---|---|---|
| Level of abstraction | Transistor-level circuit (structural) | VHDL code (behavioral) |
| Floating-point precision | Single precision (32 bit) | Half precision (16 bit) |
| Sequential circuit type | Asynchronous | Synchronous |
| Rounding unit | Absent | Present |

Noche (2003) aimed to design a relatively complex asynchronous circuit. While typical floating-point arithmetic units performed only one or two operations, he wanted to design one that performed addition, multiplication, and division. To do so within the constraints of the computing power available, the design had to be small. Noche (2003) used Goldberg's (1990) algorithms for addition, multiplication, division, and remainder because these could be implemented using shared components. Hardware implementations for floating-point remainder exist (e.g.: Sharangpani & Girard (1994)), but these are not common. The algorithms were chosen for their small area

requirements, not for their performance. The algorithms have variable latency—the resulting completion times are data-dependent.

Not all floating-point arithmetic units implement rounding in hardware. Among those that do, not all round correctly in all cases. For example, Marcus *et al.*'s (2004) VHDL model of a single-precision floating-point adder and multiplier does not correctly round some operations. The rounding unit in the present work is intended to work correctly in all cases.

Field-programmable gate arrays (FPGAs) are suitable for accelerating deep learning inference (such as those used in computer vision and natural language processing) (Boutros *et al.*, 2025). Reduced-precision floating-point types, which have faster computations and lower energy consumption, have become increasingly popular in artificial intelligence applications because these can tolerate lower precision without a significant loss in accuracy (Dongarra *et al.*, 2024). The present work's goal is to create a model of an arithmetic unit that is synthesizable in an FPGA and uses a small floating-point format. Thus, it may be useful for deep learning applications.

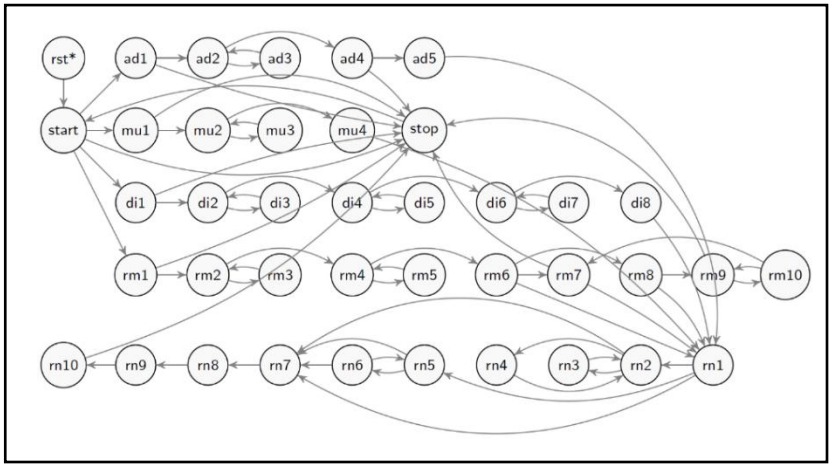


Figure 1. Simplified state diagram of the model showing the steps in the four arithmetic operations and rounding, where each state executes in one clock cycle

Figure 1 shows a simplified state diagram of the present work. The algorithms for addition, multiplication, division, and remainder are discussed by Frondozo *et al.* (2020). The rounding algorithm is discussed in detail below. The following section provides background on the IEEE standard needed for

the rounding discussion, followed by the testing approach, results, and recommendations for future work.

## 2. Methodology

### *2.1 IEEE Standard for Floating-Point Arithmetic*

In the IEEE standard for floating-point arithmetic (IEEE, 2008), a binary format known as binary16 (or half-precision) represents floating-point data using 16 bits: a 1-bit sign $S$, a 5-bit biased exponent $E$ (with a bit representation $E_5 \ldots E_1$), and a 10-bit trailing significand field $T$ (with a bit representation $f_{10} \ldots f_1$). The fraction $f$ is $T \times 2^{-10}$, i.e., the binary number $0.f_{10}f_9f_8f_7f_6f_5f_4f_3f_2f_1$. The significand is the binary number $f_{11}.f_{10}f_9f_8f_7f_6f_5f_4f_3f_2f_1$ (see §2.1.3).

The biased exponent $E$ has a bias of 15, i.e., $e = E - 15$, where $e$ is the exponent. The exponent is minimum when $E = 1$, i.e., $emin = -14$; $e$ is maximum when $E = 30$, i.e., $emax = 15$.

A triple ($S$, $e$, $f$) represents exactly one of the following data: a signed zero, a signed subnormal number, a signed normal number, a signed infinity, or a NaN (Not a Number), depending on the values of $e$ and $f$. See Table 2.

Table 2. Half-precision floating-point data (IEEE, 2008)

| $S$ | $e$ | $f$ | Represents |
|---|---|---|---|
| 0 | $-15$ | 0 | $+0$ |
| 0 | $-15$ | $\neq 0$ | $+f \times 2^{-14}$ (subnormal) |
| 0 | $-15 < e < 16$ | $f$ | $+(1 + f) \times 2^e$ (normal) |
| 0 | 16 | 0 | $+\infty$ |
| 1 | $-15$ | 0 | $-0$ |
| 1 | $-15$ | $\neq 0$ | $-f \times 2^{-14}$ (subnormal) |
| 1 | $-15 < e < 16$ | $f$ | $-(1 + f) \times 2^e$ (normal) |
| 1 | 16 | 0 | $-\infty$ |
| $S$ | 16 | $\neq 0$ | NaN |

Positive and negative zero are numerically equal. But if $x$ is a positive finite value, then $x \div +0 = +\infty$, $x \div -0 = -\infty$, $-x \div +0 = -\infty$, and $-x \div -0 = +\infty$.

### 2.1.1 Exceptions

An exception is “an event that occurs when an operation on some particular operands has no outcome suitable for every reasonable application” (IEEE, 2008, §2.1.18). There are five exceptions (invalid operation, divideByZero, overflow, underflow, inexact) and the conditions that cause them to occur are described in IEEE (2008, §7). Raising a corresponding status flag is the default handling of exceptions and is what is done in the present work. For example, for the operation $2^{-24} \times 2^{-24}$, the model rounds the answer to 0 and raises the inexact flag (because the rounded result differs from the correct answer) and the underflow flag (because the correct answer is non-zero and is too small to be represented in the format). The model has five signal outputs that show the status of each flag, and five control inputs that externally restore each flag.

### 2.1.2 NaNs

There are two kinds of NANs, signaling NaNs (which have $f_{10} = 0$) and quiet NaNs (which have $f_{10} = 1$) (IEEE, 2008, §6.2). An invalid operation where none of the operands are NaNs (such as $\infty - \infty$, $0 \times \infty$, or $0 \div 0$) results in a quiet NaN (for the present work, 0 11111 1111111111).

Signaling NaNs generally signal invalid operation exceptions when they appear as operands; quiet NaNs propagate through arithmetic operations without signaling exceptions.

If exactly one of the operands is a quiet NaN, then the result is that operand. If both operands are quiet NaNs, then the output is one of the operands (for the present work, the first operand).

If at least one of the operands is a signaling NaN, then the result is a quiet NaN, in particular, one of the signaling NaNs that has been quieted by setting $f_{10}$ to 1, leaving the remaining bits of $T$ unchanged.

### 2.1.3 Subnormals

Before the operands are processed by the present work, they are unpacked, i.e., a sixth bit (a 0) is appended to the ‘left’ of the biased exponent, and a 12th bit (a 0) and an 11th bit (the hidden bit) are appended to the ‘left’ of the fraction (creating the significand). If the operand is a subnormal, the hidden bit is set to 0. If the operand is normal, the hidden bit is set to 1. (Operands of 0, ∞, and

NaN are handled separately.) This leads to a problem in the internal representations of subnormals and normalized numbers. See Table 3.

Table 3. Problematic unpacking of subnormal numbers

| Value | Packed | Problematically unpacked |
|---|---|---|
| $2^{-16}$ | 0 00000 0100000000 | 0 000000 000100000000 |
| $2^{-15}$ | 0 00000 1000000000 | 0 000000 001000000000 |
| $2^{-14}$ | 0 00001 0000000000 | 0 000001 010000000000 |
| $2^{-13}$ | 0 00010 0000000000 | 0 000010 010000000000 |

For correct processing of subnormals and normalized numbers, the present work uses a corrected unpacking: when unpacking a subnormal, the hidden bit is set to zero and the significand is shifted left once (with the least significant bit set to zero) without changing the biased exponent. See Table 4.

Table 4. Corrected unpacking of subnormal numbers

| Value | Packed | Correctly unpacked |
|---|---|---|
| $2^{-16}$ | 0 00000 0100000000 | 0 000000 001000000000 |
| $2^{-15}$ | 0 00000 1000000000 | 0 000000 010000000000 |
| $2^{-14}$ | 0 00001 0000000000 | 0 000001 010000000000 |
| $2^{-13}$ | 0 00010 0000000000 | 0 000010 010000000000 |

The division and remainder algorithms used in the present work require normalized operands. Normalization consists of repeatedly shifting the significand once to the left and decrementing the biased exponent until the hidden bit is a 1. The present work lets subnormals be normalized by allowing the 6-bit biased exponent to have negative values. See Table 5.

Table 5. Corrected unpacking and normalization of subnormal numbers (based on Noche (2003))

| Value | Packed | Correctly unpacked and normalized |
|---|---|---|
| $2^{-16}$ | 0 00000 0100000000 | 0 111111 010000000000 |
| $2^{-15}$ | 0 00000 1000000000 | 0 000000 010000000000 |
| $2^{-14}$ | 0 00001 0000000000 | 0 000001 010000000000 |
| $2^{-13}$ | 0 00010 0000000000 | 0 000010 010000000000 |

Subnormals should have a 'packing correction' before they are placed on the output. This is done by the rounding unit as its first step, described below.

### *2.2 Rounding*

IEEE Std 754-2008 (IEEE, 2008, §4.3) specifies five ways of rounding: roundTiesToEven, which rounds to the nearest floating-point number or, if both are equally near, the one with an even least significant digit (the default); roundTiesToAway, which rounds to the nearest floating-point number or, if both are equally near, the one with the larger magnitude; roundTowardZero; roundTowardPositive; and roundTowardNegative. The present work only implements roundTiesToEven, roundTowardZero, roundTowardPositive, and roundTowardNegative. ("The rounding attribute roundTiesToAway is not required for a binary format implementation" (IEEE, 2008, §4.3.3).)

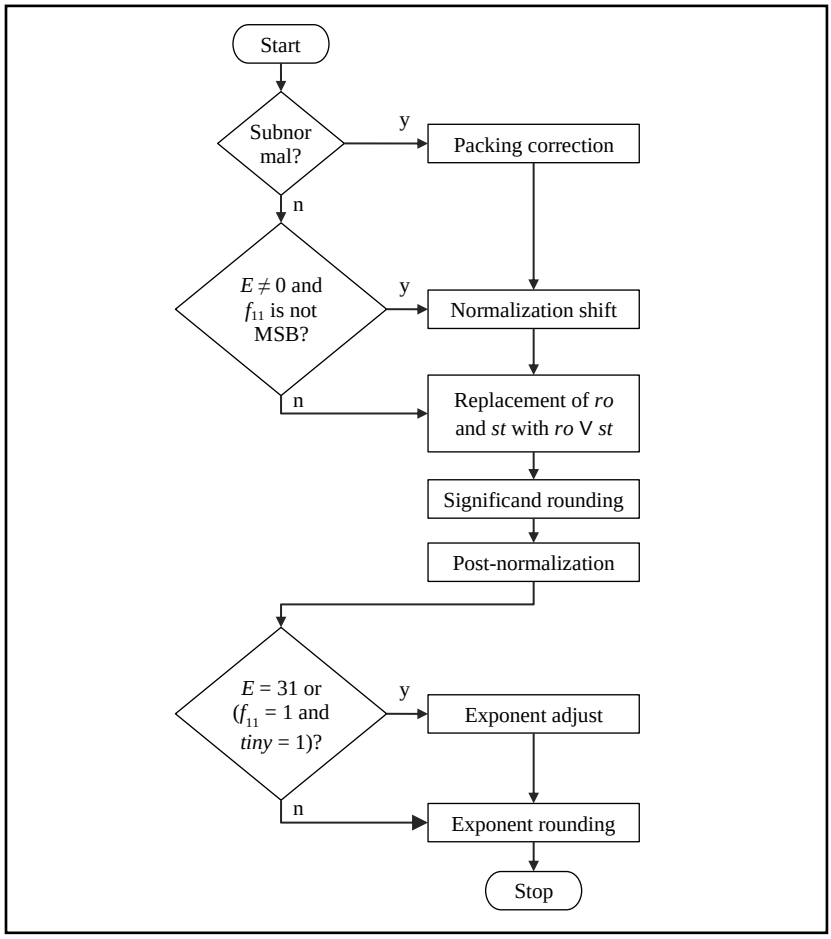


Figure 2. Simplified flowchart of the rounding algorithm

The standard also specifies that the result of an arithmetic operation be the same as if it were computed exactly and then rounded using the current mode. Using three extra bits in the operations will accomplish this: a guard bit *gu*, a round bit *ro*, and a sticky bit *st*. Whenever the significand is shifted left, *st* is made 0. Whenever the significand is shifted right, *st* after shifting is the value of *ro* ∨ *st* before shifting (Goldberg, 1990, p. I-18).

Figure 2 shows a simplified flowchart of the rounding algorithm. Rounding is illustrated here through some results input to the rounding unit. See Table 6. Steps 1 to 6 are from Even and Paul (2000).

Step 0: Packing correction. Subnormals undergo packing correction as described above: If the absolute value of the result is less than $2^{\text{emin}}$ (i.e., $2^{-14}$), then the biased exponent is unchanged, and the significand is shifted right once. See Table 7. M35 and M40 have negative exponents (110010 is −14, 111111 is −1) and so underwent packing correction; M48 has a positive exponent (101110 is 46) and so did not undergo packing correction.

Table 6. Rounding: Some results input to the rounding unit

| Label | *S* | $E_6,E_5,\ldots,E_1$ | $f_{12},f_{11},f_{10},\ldots,f_1,gu,ro,st$ |
|---|---|---|---|
| A29 | 0 | 000000 | 000000000100000 |
| A53 | 0 | 000000 | 100000000000000 |
| A65 | 0 | 011110 | 111111111110000 |
| M35 | 1 | 110010 | 000000000010000 |
| M40 | 0 | 111111 | 011111111110000 |
| M41 | 0 | 001110 | 011111111111111 |
| M48 | 0 | 101110 | 011111111110001 |
| R48 | 1 | 010001 | 000100000000000 |

Table 7. Rounding: After packing correction

| Label | *S* | $E_6,E_5,\ldots,E_1$ | $f_{12},f_{11},f_{10},\ldots,f_1,gu,ro,st$ |
|---|---|---|---|
| A29 | 0 | 000000 | 000000000010000 |
| A53 | 0 | 000000 | 100000000000000 |
| A65 | 0 | 011110 | 111111111110000 |
| M35 | 1 | 110010 | 000000000001000 |
| M40 | 0 | 111111 | 001111111111000 |
| M41 | 0 | 001110 | 011111111111111 |
| M48 | 0 | 101110 | 011111111110001 |
| R48 | 1 | 010001 | 000100000000000 |

Step 1. Normalization shift. If correction was performed in the previous step, then the *tiny* flag is set, and the significand is shifted, and the biased exponent

correspondingly incremented or decremented until $E = 0$. Otherwise, if $E \neq 0$, then the significand is shifted, and the biased exponent correspondingly incremented or decremented until $f_{11}$ holds the most significant bit. The $ovf_1$ flag is set if the absolute value of the result is greater than or equal to $2^{emax + 1}$ (i.e., $2^{16}$). See Table 8.

Table 8. Rounding: After normalization shift

| Label | $S$ | $E_6,E_5,\ldots,E_1$ | $f_{12},f_{11},f_{10},\ldots,f_1,gu,ro,st$ | $tiny$ | $ovf_1$ |
|---|---|---|---|---|---|
| A29 | 0 | 000000 | 000000000010000 | 1 | 0 |
| A53 | 0 | 000000 | 100000000000000 | 0 | 0 |
| A65 | 0 | 011111 | 011111111111000 | 0 | 1 |
| M35 | 1 | 000000 | 000000000000001 | 1 | 0 |
| M40 | 0 | 000000 | 000111111111100 | 1 | 0 |
| M41 | 0 | 001110 | 011111111111111 | 0 | 0 |
| M48 | 0 | 101110 | 011111111110001 | 0 | 1 |
| R48 | 1 | 001111 | 010000000000000 | 0 | 0 |

Step 2. Replacement of *ro* and *st* with *ro* ∨ *st*. The round and the sticky bits are then replaced with a bit that is their logical or. See Table 9.

Table 9. Rounding: Replacement of *ro* and *st* with *ro* ∨ *st*

| Label | $S$ | $E_6,E_5,\ldots,E_1$ | $f_{12},f_{11},f_{10},\ldots,f_1,gu,ro \vee st$ |
|---|---|---|---|
| A29 | 0 | 000000 | 000000000010000 |
| A53 | 0 | 000000 | 100000000000000 |
| A65 | 0 | 011111 | 011111111111000 |
| M35 | 1 | 000000 | 000000000000001 |
| M40 | 0 | 000000 | 000111111111110 |
| M41 | 0 | 001110 | 011111111111111 |
| M48 | 0 | 101110 | 011111111110001 |
| R48 | 1 | 001111 | 010000000000000 |

Step 3. Significand rounding. Bits *gu* and *ro* ∨ *st* are set to 0, and the significand $f_{12},\ldots,f_1$ is either unchanged or incremented depending on $f_1$, *gu*, *ro* ∨ *st*, *S*, and the rounding mode. If rounding the significand results in $f_{12},f_{11},\ldots,f_1 = 10\ldots0$, then the *sig_ovf* flag is set. If the significand or the *gu* or *ro* ∨ *st* bits change after rounding, then the *sig_inx* flag is set. See Table 10.

Table 10. Rounding: Significand rounding

| Label | $S$ | $E_6,E_5,\ldots,E_1$ | $f_{12},f_{11},f_{10},\ldots,f_1,gu,ro \vee st$ | *sig_ovf* | *sig_inx* |
|---|---|---|---|---|---|
| A29 | 0 | 000000 | 00000000001000 | 0 | 0 |
| A53 | 0 | 000000 | 10000000000000 | 1 | 0 |
| A65 | 0 | 011111 | 01111111111100 | 0 | 0 |
| M35 | 1 | 000000 | 00000000000000 | 0 | 1 |
| M40 | 0 | 000000 | 00100000000000 | 0 | 1 |
| M41 | 0 | 001110 | 10000000000000 | 1 | 1 |
| M48 | 0 | 101110 | 01111111111000 | 0 | 1 |
| R48 | 1 | 001111 | 01000000000000 | 0 | 0 |

Table 11. Rounding: Post-normalization

| Label | $S$ | $E_6,E_5,\ldots,E_1$ | $f_{11},f_{10},\ldots,f_1$ |
|---|---|---|---|
| A29 | 0 | 000000 | 00000000010 |
| A53 | 0 | 000001 | 10000000000 |
| A65 | 0 | 011111 | 11111111111 |
| M35 | 1 | 000000 | 00000000000 |
| M40 | 0 | 000000 | 01000000000 |
| M41 | 0 | 001111 | 10000000000 |
| M48 | 0 | 101110 | 11111111110 |
| R48 | 1 | 001111 | 10000000000 |

Step 4. Post-normalization. Bits $f_{12}$, *gu*, and *ro* ∨ *st* are removed. If the *sig_ovf* flag was set in the previous step, then the exponent is incremented and $f_{11}$ is set to 1. See Table 11.

Step 5. Exponent adjust. If $E = 31$ (i.e., $e = emax + 1 = 16$), then the $ovf_2$ flag is set. If $f_{11} = 1$ and *tiny* was set, then the subnormal has become normalized and the biased exponent must be incremented (i.e., set to $E = 1$). See Table 12.

Table 12. Rounding: Exponent adjust

| Label | $S$ | $E_6,E_5,\ldots,E_1$ | $f_{11},f_{10},\ldots,f_1$ | $ovf_2$ |
|---|---|---|---|---|
| A29 | 0 | 000000 | 00000000010 | 0 |
| A53 | 0 | 000001 | 10000000000 | 0 |
| A65 | 0 | 011111 | 11111111111 | 1 |
| M35 | 1 | 000000 | 00000000000 | 0 |
| M40 | 0 | 000000 | 01000000000 | 0 |
| M41 | 0 | 001111 | 10000000000 | 0 |
| M48 | 0 | 101110 | 11111111110 | 0 |
| R48 | 1 | 001111 | 10000000000 | 0 |

Step 6. Exponent rounding. Bits $E_6$ and $f_{11}$ are removed. If the $ovf_1$ and $ovf_2$ flags were not set, then the biased exponent and the significand are unchanged.

Otherwise, either the biased exponent is set to $E = 30$ and the significand is set to $f_{10},\ldots,f_1 = 1\ldots1$, or the biased exponent is set to $E = 31$ and the significand is set to $f_{10},\ldots,f_1 = 0\ldots0$, depending on the rounding mode and $S$. The overflow, underflow, and inexact exceptions are signaled by the *ovr*, *und*, and *inx* flags, respectively, where $ovr = ovf_1 \vee ovf_2$, $und = tiny \wedge sig\_inx$, and $inx = sig\_inx \vee ovr$. See Table 13.

Table 13. Rounding: Exponent rounding

| Label | $S$ | $E_5,\ldots,E_1$ | $f_{10},\ldots,f_1$ | *und* | *ovr* | *inx* |
|---|---|---|---|---|---|---|
| A29 | 0 | 00000 | 0000000010 | 0 | 0 | 0 |
| A53 | 0 | 00001 | 0000000000 | 0 | 0 | 0 |
| A65 | 0 | 11111 | 0000000000 | 0 | 1 | 1 |
| M35 | 1 | 00000 | 0000000000 | 1 | 0 | 1 |
| M40 | 0 | 00000 | 1000000000 | 1 | 0 | 1 |
| M41 | 0 | 01111 | 0000000000 | 0 | 0 | 1 |
| M48 | 0 | 11111 | 0000000000 | 0 | 1 | 1 |
| R48 | 1 | 01111 | 0000000000 | 0 | 0 | 0 |

After the values in Table 6 are input to the rounding unit, the rounding unit outputs the values in Table 13.

*2.3 Research Approach and Data Collection*

In this quantitative, experimental research study, an existing design was modified and implemented in a different way. The new work was tested using inputs that were similar to those of the previous one, and the outputs were compared to verify that the new work behaves similarly to the old one.

The VHDL code was simulated using Lattice Diamond 3.10 design software and tested using Active-HDL 10.3 FPGA Design and Simulation Lattice Edition II. Of the 248 test vectors of Noche (2003), two had no half-precision equivalent. The rest were modified for half-precision, yielding a set of 246 test vectors. These test vectors cover the four operations (addition, multiplication, division, remainder), the five floating-point data (zeroes, subnormals, normals, infinities, NaNs (signaling and quiet)), the five exceptions (invalid operation, divideByZero, overflow, underflow, inexact), and the four rounding modes (roundTiesToEven, roundTowardPositive, roundTowardNegative, roundTowardZero). The test vectors were chosen to test the model's functionality and to characterize its performance. Verdonk *et al*. (2001) is the source of 174 of the 246 test vectors used: 50 of the 68 addition vectors, 39 of the 51 multiplication vectors, 42 of the 53 division

vectors, and 43 of the 74 remainder vectors. The remaining 72 test vectors were created by the authors.

## 3. Results and Discussion

The variable-latency algorithms used in the present work are the same as those of Noche (2003). Noche's (2003) work is asynchronous, so its completion times are in nanoseconds. The present work is synchronous with no set clock period, so its times are in clock cycles.

Noche and Araneta (2007) show that, in their work, addition times vary linearly with $x$ (the absolute difference of the exponents of the two operands), multiplication times vary linearly with $n_x$ (the number of 1's in the significand of the first operand), division times vary linearly with $n_z$ (the number of 1's in the unrounded significand of the result) and with $s$ (the number of shifts to normalize both operands), and remainder times vary linearly with $z$ (the number of shifts taking into account the difference in operand exponents and significands being multiples of each other) and with $s$.

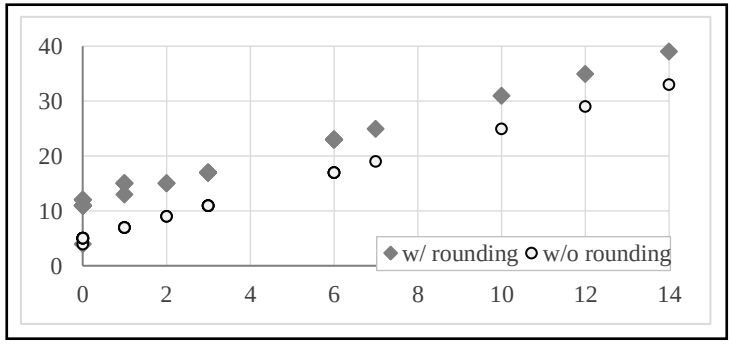


Figure 3. Addition completion times (ordinary cases) in clock cycles versus $x$

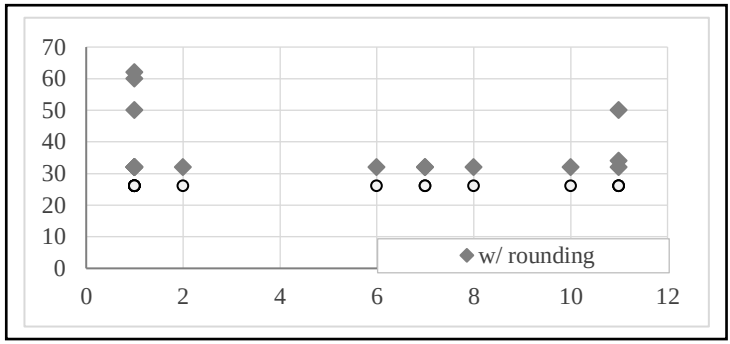


Figure 4. Multiplication completion times (ordinary cases) in clock cycles versus $n_x$

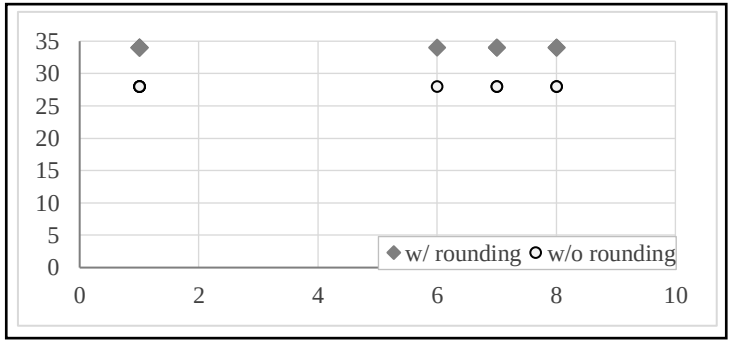


Figure 5. Division completion times (normal operands) in clock cycles versus $n_{z\backslash}$

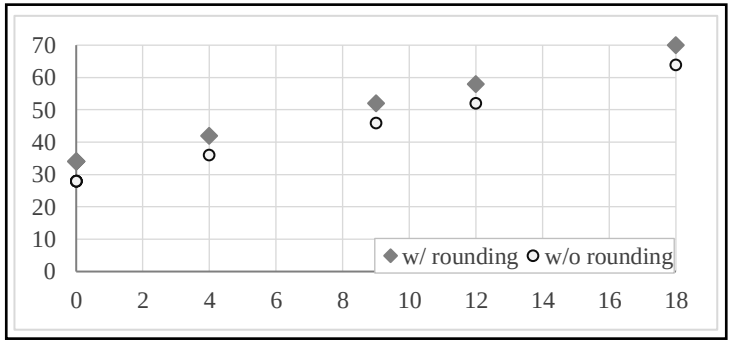


Figure 6. Division completion times (ordinary cases with one 1 in the unrounded significand of the result) in clock cycles versus $s$

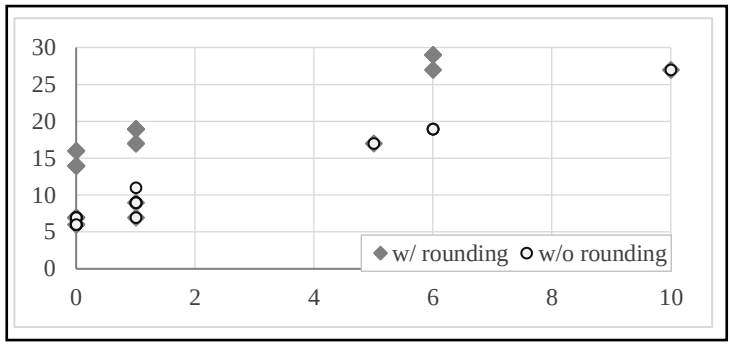


Figure 7. Remainder completion times (normal operands with the first having an exponent greater than or equal to that of the second) in clock cycles versus *z*

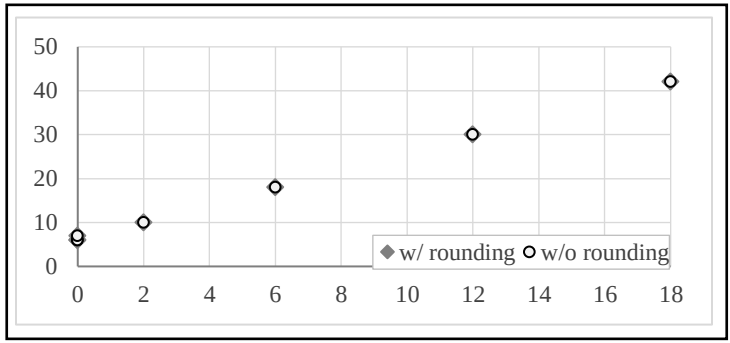


Figure 8. Remainder completion times (ordinary cases with equal operands) in clock cycles versus *s*

Figures 3 to 8 show the times (with and without rounding) for selected test vectors (Noche & Araneta's (2007) times exclude rounding). Table 14 shows the lines that best fit the data using the least squares method. The linear relationships found by Noche and Araneta (2007) are also found in the present work except those involving the number of 1's in significands (Figures 4 and 5). Noche and Araneta's (2007) work used variable-latency asynchronous circuits, in particular, adders that finish earlier if there are fewer 1's in the significand. The adders in the present work follow a clock and so have fixed latency.

The completion times with rounding varied from 1 to 39 cycles for addition, 1 to 62 for multiplication, 1 to 102 for division, and 1 to 60 for remainder. Typical latencies (in other work) are 3 cycles for addition, 5 for multiplication,

and 15 for division (Boldo *et al*., 2023), but it is unclear if these values include rounding.

Table 14. Lines that best fit the data using the least squares method

| Operation | With rounding | Without rounding |
|---|---|---|
| Addition | $y = 1.999x + 11.07$ | $y = 2.008x + 4.94$ |
| Multiplication | $y = -0.676n_x + 41.38$ | $y = 26$ |
| Division 1 | $y = 34$ | $y = 28$ |
| Division 2 | $y = 2s + 34$ | $y = 2s + 28$ |
| Remainder 1 | $y = 2.132z + 11.51$ | $y = 2.071z + 6.57$ |
| Remainder 2 | $y = 1.982s + 6.24$ | $y = 1.982s + 6.24$ |

Correct results were obtained, including the setting of the appropriate flags, in all but two cases: $2^{-24} \times (2^{16} - 2^5)$ and $(2^{16} - 2^5) \times 2^{-24}$ (the product of the largest normal and the smallest subnormal). The correct result in both cases is $(2^{-8} - 2^{-19})$ (0 00110 1111111111) but the result given by the present work in both cases is (0 00110 1111000000), a relative error of 63/2047 ≈ 3%.

In the present work, the 11-bit significand has a $12^{th}$ bit appended to the left to avoid data loss due to overflows. The product of two (12-bit) significands is stored using 24 bits with three bits appended to the right to ensure correct rounding (Goldberg, 1990) so in effect the product is stored using 27 bits. It turns out that this design decision does not properly consider subnormals. When intermediate values in the 27-bit storage are shifted to the right in these two cases, then shifted back to the left later, the six rightmost bits are lost. Using a 33-bit storage for this intermediate product might fix this problem, but this will result in slightly larger area requirements, slightly higher power consumption, and possibly slightly longer completion times.

## 4. Conclusion and Recommendation

A VHDL model of an IEEE Std 754-2008 half-precision floating-point arithmetic unit that handles addition, multiplication, division, and remainder (all with rounding) was designed and tested. It operates correctly except for some cases where a very large number and a very small number are multiplied.

Castro and Tandog (2025) modified this model twice, creating a single-precision model and a double-precision (64-bit) model. The three models were implemented on an FPGA and were verified to have correct outputs to similar

sets of test inputs (except for the cases mentioned earlier). Thus, the model described in this paper is scalable and synthesizable.

One possible direction for future work is modifying Castro and Tandog's (2025) units so that all operations are correctly rounded, then comparing the differences in area requirements, power consumption, and completion times. Another is the use of a different floating-point type, such as TensorFloat-32, Bfloat16, FP8 E4M3, or FP8 E5M2 (Dongarra *et al.*, 2024).

## 5. Acknowledgement

Ateneo de Naga University, through its Office of the Academic Vice President's Faculty and Staff Development Program, provided funding for the authors to present a preliminary version of this paper at the 1st Asia Pacific Regional Conference of the International Association of Jesuit Engineering Schools, held at Ateneo de Davao University, Davao City, Philippines, on December 1–2, 2023.

## 6. References

Boldo, S., Jeannerod, C.-P., Melquiond, G., & Muller, J.-M. (2023). Floating-point arithmetic. *Acta Numerica*, *32*, 203–290. https://doi.org/10.1017/S0962492922000101

Boutros, A., Arora, A., & Betz, V. (2025). Field-programmable gate array architecture for deep learning: Survey and future directions. *Proceedings of the IEEE*, *113*(7), 613–639. https://doi.org/10.1109/JPROC.2025.3623023

Castro, C.A.A., & Tandog, M.B.D. (2025). FPGA implementation of an IEEE 754-2019 floating-point arithmetic unit (B.S. Thesis). Department of Electronics and Computer Engineering, Ateneo de Naga University, Naga City, Philippines.

Dongarra, J., Gunnels, J., Bayraktar, H., Haidar, A., & Ernst, D. (2024). Hardware trends impacting floating-point computations in scientific applications. https://doi.org/10.48550/arXiv.2411.12090

Even, G., & Paul, W. (1997). On the design of IEEE compliant floating point units. In Proceedings of the 13th IEEE Symposium on Computer Arithmetic (pp. 54–63). Asilomar, CA, USA: IEEE. https://doi.org/10.1109/ARITH.1997.614879

Frondozo, G.A., Odiame, K.C., & Respeto, D.A.T. (2020). VHDL model of a half-precision floating-point arithmetic unit with remainder and rounding (B.S. thesis). Department of Electronics and Computer Engineering, Ateneo de Naga University, Naga City, Philippines.

Goldberg, D. (1990). Computer arithmetic. In: D.A. Patterson & J.L. Hennessy, Computer architecture: A quantitative approach (Appendix I). San Mateo, CA: Morgan Kaufmann Publishers.

IEEE. (1985). IEEE Standard for Binary Floating-Point Arithmetic. Retrieved from https://doi.org/10.1109/IEEESTD.1985.82928
IEEE. (2008). IEEE Standard for Floating-Point Arithmetic. Retrieved from https://doi.org/10.1109/IEEESTD.2008.4610935

Marcus, G., Hinojosa, P., Avila, A., & Nolazco-Flores, J. (2004). A fully synthesizable single-precision, floating-point adder/substractor and multiplier in VHDL for general and educational use. In Proceedings of the 5th IEEE International Caracas Conference on Devices, Circuits and Systems, Punta Cana, Dominican Republic: IEEE. https://doi.org/10.1109/ICCDCS.2004.1393405

Noche, J.R. (2003). An asynchronous single-precision floating-point arithmetic unit (M.S. thesis). Department of Electrical and Electronics Engineering, University of the Philippines, Diliman, Philippines.

Noche, J.R., & Araneta, J.C. (2007). An asynchronous IEEE floating-point arithmetic unit. *Science Diliman, 19*(2), 12–22. https://saliksik.upd.edu.ph/journal-issue/SD-12-25-001

Sharangpani, H., & Girard, L. (1994). Floating point remainder generator for a math processor (U.S. Patent No. 5,357,455). U.S. Patent and Trademark Office.

Sheikh, B.R., & Manohar, R. (2010). An operand-optimized asynchronous IEEE 754 double-precision floating-point adder. In 16th IEEE Symposium on Asynchronous Circuits and Systems, Grenoble, France, 151–162. https://doi.org/10.1109/ASYNC.2010.24

Srivastava, P., Chung, E., & Ozana, S. (2020). Asynchronous floating-point adders and communication protocols: A survey. *Electronics, 9*(10), 1687. https://doi.org/10.3390/electronics9101687

Verdonk, B., Cuyt, A., & Verschaeren, D. (2001). A precision and range independent tool for testing floating-point arithmetic I: Basic operations, square root and remainder. *ACM Transactions on Mathematical Software, 27*(1), 92–118. https://doi.org/10.1145/382043.382404