POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

VLSI architectures optimized for the computation of floating-point transcendental functions

Supervisors

Prof. Maurizio MARTINA

Prof. Guido MASERA

Dr. Walid WALID

Candidate

Christian VESPO

2022/2023

Abstract

Nowadays the computation of trigonometric functions has great significance in various scientific areas, such as robotics, signal and image processing, 3-D graphics, and communication systems. But usually, it is performed with software routine or with the architecture of the processor using floating-point instructions. This may present long latency and slow down the program execution by spending the majority of the time in long trigonometric computation. Improvement in this can be achieved by using a dedicated unit for the calculation of trigonometric identities. Thus, this thesis aims to implement this computation with dedicated hardware to have high performances in terms of frequency, clock cycles, and instructions using the algorithm known as COordinate Rotation DIgital Computer (CORDIC). It is an iterative algorithm with which it is possible to compute trigonometric functions such as sine, cosine, hyperbolic sine, hyperbolic cosine, exponential and logarithmic functions, and multiplication and division operations. By taking initial the argument of the function as input, along with two initial values that are already established by the algorithm for each function. The algorithm computes the result in the same way, independently of the type of function selected. The operations performed in an iteration depend on the sign of the argument of the function chosen, and with each iteration, increasingly accurate values for the functions are produced and the argument is updated. The algorithm terminates in the case in which the argument reaches zero. However, it may take many iterations to reach this value, and this could be a problem depending on the task at hand. Thus an optimum number of iterations is required. For this reason, the number of iterations chosen for all the implementations developed in this thesis is 5, intended as a good compromise between the accuracy of the results and the latency needed for the computation. In this work, the binary format IEEE-754 single-precision floating-point is used for all the implementations. The initial objective consists of implementing an architecture for CORDIC that can operate at the frequency of the order of 1GHz to be compatible with modern processor cores. To reach this target, a pipeline stage has been gradually added to the starting architecture in the area identified as a critical path, generating in this way different versions of the same architecture. For each version the area, frequency, and latency have been computed, using the logic synthesis tool Synopsys Design Vision, and the simulation tool Modelsim. To enhance the latency of the iterative approach, a different version of the architecture is proposed, and it is an unrolled version of the basic CORDIC algorithm. To test the proposed design in a real scenario a RISC-V processor core is utilized. For this reason, PULPino, an open-source single-core microcontroller system from the literature, has been chosen based on 32-bit RISC-V cores. The

same trigonometric functions have been computed on PULPino with and without CORDIC architecture, computing the number of clock cycles and the number of instructions in both cases. The results demonstrated that increasing the number of trigonometric functions to calculate, PULPino using CORDIC architectures takes fewer clock cycles and instructions than using its computation resources, but the cost is paid in terms of area.

Summary

Nowadays the computation of trigonometric functions has great importance in various scientific areas, such as communication systems, robotics, digital signal processing, software radio, 3-D graphics. But usually, it is performed with the processor using floating-point instructions or in software using routines. But this can slow down the program execution, because the processor could spend majority of the time in long trigonometric computations. For this reason the need emerged to assign the calculation of trigonometric identities to a dedicated hardware unit. Thus, this thesis work aims to implement this computation with dedicated hardware to reach high performances in terms of clock cycles, instructions, and frequency using the algorithm known as COordinate Rotation DIgital Computer (CORDIC). It is an iterative algorithm with which it is possible to compute trigonometric functions such as sine and cosine, exponential and logarithmic functions, hyperbolic functions such as hyperbolic sine and hyperbolic cosine, and multiplication and division operations. This algorithm operates in two ways, known as Rotation mode, that is the mode chosen in this thesis, and Vectoring mode, but for both the general behavior is the same: at first, the initial argument of the function is taken as input, along with two initial values that are already established by the algorithm for each function. The operations performed in an iteration depend on the direction of the rotation d_i, which can have as values either +1 or -1. This direction in Rotation mode is equal to the sign of the value in the iteration taken by the argument of the function chosen, instead in Vectoring mode this direction is equal to the opposite value of the sign of the product of the values in the iteration taken by the other two functions. At each iteration, increasingly accurate values for the functions are produced and the argument is updated. The algorithm terminates in the case in which the argument reaches zero or a value close to zero. However, it may take many iterations to reach this value, becoming a possible problem for the application in which it works. Thus an optimum number of iterations is required. In this thesis, the number of iterations chosen for all the implementations developed is 5, and it constitutes a good compromise between the accuracy of the results and the latency needed for the computation. Moreover for all the implementations is used the binary format IEEE-754 single-precision

floating-point format. The initial objective consists of implementing an architecture for CORDIC that can work at the frequency of the order of 1 GHz in order to be compatible with modern processor cores. Thus, the architectures in the literature were studied, and among them one chosen that focused on working at high frequency and minimizing the area. But this architecture was appropriate for two complement format and fixed-point format. Therefore it was adapted to the floating-point format, and the most important change was to replace the present asynchronous binary adder with an asynchronous floating-point adder. So the latter was implemented and then was subsequently tested with the UVM testbench. Then the maximum clock frequency and the area of this floating-point CORDIC architecture were measured using the logic synthesis tool Synopsys Design Vision, and the maximum frequency reached by this architecture was equal to 251.26MHz. From the critical path analysis, it was identified that the floating-point adder was the component that introduced the largest contribution to the delay. Thus the pipeline stages were gradually introduced within the floating-point adder, and the CORDIC architecture constituted by floating-point adders with 4 pipeline stages reached a frequency equal to 641.03MHz. In addition, soliciting Synopsys to make optimizations on the architecture, the latter could achieve a frequency of 970.87MHz. But this architecture has a defect, it consists of a high latency: for this type of architecture, pipelined floating-point adders are not used to their full efficiency, in other words, it is not possible to wait for N clock cycles as many as the number of pipeline stages introduced, to compute the sums in the first iteration and then begin the computation of the sums of subsequent iterations each clock cycle. This is because a given data dependency is present, since each iteration starts the computations with the values of the previous iteration. So in this case the pipelined floating-point adders are useful only for reducing the critical path of the circuit, and not for doing parallel operations. So, as suggested in the literature, an unrolled version of the CORDIC architecture was implemented, obtained by instantiating the previous CORDIC one, making some changes, a number of times equal to the number of iterations chosen, which is five. The unrolled CORDIC architecture with four pipeline stages reached a maximum clock frequency of 621.12MHz, and with Synopsys optimization, this frequency reached a value equal to 813MHz. Moreover, this architecture has a latency smaller than the CORDIC one by an amount equal to 7 clock cycles, and using this type of architecture it is possible to start the computation of new trigonometric functions at each clock cycle. But using the unrolled version has a cost in terms of area, since its area is 5 times larger than the cordic version. To test the proposed design in a real scenario, a RISC-V processor core is utilized. For this reason, PULPino, an open-source single-core microcontroller system from the literature, has been chosed based on 32-bit RISC-V cores. Thus PULPino was compiled, and then an APB bus was created to connect the CORDIC architectures to the system.

Then the same trigonometric functions have been computed on PULPino with and without CORDIC architecture, computing the number of clock cycles and the number of instructions in both cases. In particular, internally PULPino has performance counters, that compute the event that is specified, such as clock cycles, instructions, jumps, and branches. To compute the number of clock cycles and instructions, it is sufficient to access these counters, and starting from zero begin counting the clock cycles and instructions at the beginning of the computation of functions, then stop these counters as soon as the functions have been computed, thus reading the values of these counters. From the results obtained, using the two types of architectures implemented PULPino employs the same number of instructions, which is approximately half the number of instructions employed to use the floating-point mathematical functions internal to PULPino. Regarding the clock cycles measured on PULPino, performance counters were not used during the calculation of functions with the CORDIC architectures, because the number read by the counter also counts the clock cycles spent to communicate with the peripheral. Using the internal mathematical functions on PULPino, the calculation of five trigonometric functions takes 245 clock cycles. Instead, using Unrolled CORDIC architecture for the calculation of the first function requires 30 clock cycles, and then other 4 clock cycles to compute the remaining four functions. Therefore the Unrolled CORDIC architecture can be considered a valid alternative to PULPino's mathematical functions.

Acknowledgements

I wish to thank Prof. Maurizio Martina for his availability, and for the encouragement and precious advice received during the whole thesis work.
I would also like to thank Prof. Guido Masera for his availability and for the concepts I learned during his explanations, which have been very useful for the thesis work.
I would also to thank Ing. Walid Walid for all his support, encouragement and advice I received. It has been an honour for me to work with you.
I would also to thank the VLSI lab for welcoming me and making me feel comfortable during my thesis work.
I wish to thank my parents, my brothers Vincenzo and Marco, and my grandparents for their support, constant presence and encouragements. All of you are my strength.
Finally, I would like to thank all my friends for making these years of college beautiful and unforgettable.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays the computation of trigonometric functions has great relevance in various places such as digital signal processing, robot control, software radio, math processors, navigation systems [1], wireless communications [2]. Normally their values are calculated with software routine or using the units of calculation of the processor, but this negatively affects the performance in terms of speed of the whole architecture, just as reported in [3]. In fact, to compute trigonometric functions the processor can spend the majority of the time in long computations, causing a slowdown of the program execution. Therefore it is very important that it works together with a hardware architecture dedicated to the calculation of trigonometric identities, to reach higher performance.

This thesis aims to design this hardware architecture. Generally, various methods exist in literature to compute hardware functions such as sine, cosine, hyperbolic sine, hyperbolic cosine, exponential, and logarithmic functions. Some methods use a lookup table(LUT) or memory approach [4], others use techniques such as Taylor's series, but most of these use COordinate Rotation DIgital Computer (CORDIC) algorithm [1, 2, 3, 5, 6, 7, 8, 9]. Lookup table or memory approach consists in storing the results of the chosen function into a memory. Therefore at each entrance angle corresponds a value of the memory. The hardware implementations based on this approach are simple and fast, but they have problems related to accuracy and area efficiency. With small memory, the results are inaccurate, although increasing the size of the memory their values are more accurate, but it also makes the hardware implementation inefficient in terms of the area. The technique based on Taylor's series consists of calculating the functions using the related Taylor's series. For example, the sine function can be computed as shown in the equation 1.1

$$sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} ... \tag{1.1}$$

where x denotes the angle. To implement this method an adder, subtractor,

multiplier, and divider is required. To have higher accuracy, both higher factorial terms and higher powers are required. Therefore this method is slow and requires a large area 1-4.

CORDIC is an iterative algorithm. Starting from the argument of the function and two initial values, that are already established by the algorithm for each function, it computes at each iteration a more accurate value of the function. It is explained in detail in 2.3. Its advantage over other methods is that it uses adders, subtractors, shifters, small LUT, and not multipliers and dividers. In this way, the architecture is fast and efficient in area[1]. For these good reasons, the algorithm chosen in this thesis is CORDIC. The binary format IEEE-754 single-precision floating-point is used for all the implementations. The initial objective consists of implementing an architecture for CORDIC that can operate at the frequency of the order of 1GHz to be compatible with modern processor cores. Thus CORDIC architectures found in the literature [1, 2, 5, 7] have been very helpful in figuring out which type of implementation was most appropriate for working at high frequency. These architectures presented had significant latency, thus more solutions have been explored [6], [7]. This thesis offers CORDIC architectures that can work at the frequency of the order of 1GHz and have low latency.
Moreover, these architectures are tested on PULPino [10], an open-source single-core microcontroller system based on 32-bit RISC-V cores.
The remaining of this thesis is organized as follows:

1. Chapter 2 presents the Background: Here floating point format, floating point addition, and the CORDIC algorithm are explained in detail. Moreover, all the scientific articles studied are summarized, and the differences among the architecture present in the literature will be presented.

2. In chapter 3 the proposal architectures will be explained. They comprise the pipelined CORDIC and the unrolled CORDIC. The chapter also details the implementation strategies of the proposed CORDIC.

3. In chapter 4, a RISC-V core called the PULPino platform from the literature is presented. Including all the steps to build PULPino on the server; how the proposed architecture is connected to PULPino how it is executed and how the results about clock cycles and number of instructions are computed.

4. In chapter 5 the results are presented for all the versions of the CORDIC implemented. Also, a comparison with state-of-the-art is provided.

5. In chapter 6 the conclusions of the whole work are given. In addition, some future works on this topic are also proposed.

Chapter 2

Background

2.1 Floating point format

IEEE-754 single-precision floating-point format has 32 bits, divided into sign, exponent, and mantissa. The sign is the Most Significant Bit (MSB), and it is positive if its value is equal to zero logic, otherwise, it is negative. The exponent is unsigned and consists of 8 bits, ranging from position 30 to 24. Its minimum decimal value is zero, its maximum decimal value is 255. The mantissa is formed by 23 bits, ranging from position 23 to 0. Its decimal value is greater than or equal to zero and it is lower than one. A representation of this format is given in figure 2.1.

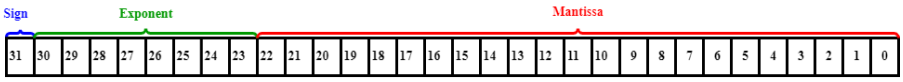

Figure 2.1: IEEE-754 single-precision floating-point format

The decimal value corresponding to the binary representation is shown by the following expression

$$decimal\ value = -1^{Sign} \times 2^{Exp-Exp\_bias} \times (1 + \sum_{n=1}^{23} b_{23-n}\ 2^{-n}) \tag{2.1}$$

of which

$$Exp\_bias = 2^{Num\_bits\_Exp\ -1} - 1 = 2^{8-1} - 1 = 127 \tag{2.2}$$

For example, the number 01000000101000000000000000000000 has

- sign bit equal to zero logic, therefore the sign is positive
- the decimal value of the exponent is 129

- the decimal value of the mantissa is equal to 0.25

Using the equation 2.1, this binary value is equal to 5.
Instead, to convert a decimal number to the floating-point format, it is necessary to follow these steps:

1. Convert decimal numbers in binary fixed-point format
2. Represent the number obtained in binary scientific notation
3. Convert the number in IEEE-754 single-precision floating-point format, making sure both to add the exponent of the number with the exponent bias to get the new exponent and to consider that the new mantissa is composed of all the bits that are to the right of the fixed point.

As an example, the floating point representation of the number 88.6 is shown below

$$88.6 \xrightarrow{fixed-point} 1011000.10011001100110011$$

$$1011000.10011001100110011 \xrightarrow{binary\ scientific\ notation} 1.01100010011001100110011 \times 2^6$$

$$1.01100010011001100110011 \times 2^6 \xrightarrow{IEEE-754} 01000010101100010011001100110011$$

2.2 Floating point addition

To add two floating point numbers, these steps must be followed:

1. At each mantissa a bit equal to one must be added in the MSB position, bringing their size to 24 bits.
2. Both numbers must have the same exponent. Otherwise, for the number that has a lower exponent, it needs to make a shift right of its mantissa of a quantity equal to the magnitude of the difference between the two exponents. In this case, the exponent of the final result is equal to the value of the bigger exponent, but it can change in the next steps.
3. If the two numbers have the same sign, of course, is the case of the addition, and the sign of the final result is equal to the sign of one of the two operands. Therefore the two mantissas can be added. If the carry_out of the sum is equal to one, the sum is shifted to the right by one position and the exponent of the final result is increased by one.
4. If the two numbers have different signs, the mantissa of the positive number is subtracted from the mantissa of the negative number. The sign of the final result is equal to the opposite value of the carry-out. If the value of carry-out

is zero, it is need to change the sum with its opposite value. In a binary system, this means changing each bit of the sum with the bit of the opposite value and then adding one to this modified sum. Moreover, if the MSB of the sum is equal to zero, it is needed both to decrease one the value of the exponent of the final result and to shift to the left the sum until the MSB bit of the sum is equal to 1.

5. To get the final result, it needs to concatenate the MSB, exponent, and the range of bits of the sum between 22 and 0 positions computed in the previous steps.

For a better understanding of these steps, an algorithmic state machine(ASM) chart is represented in figure 2.2

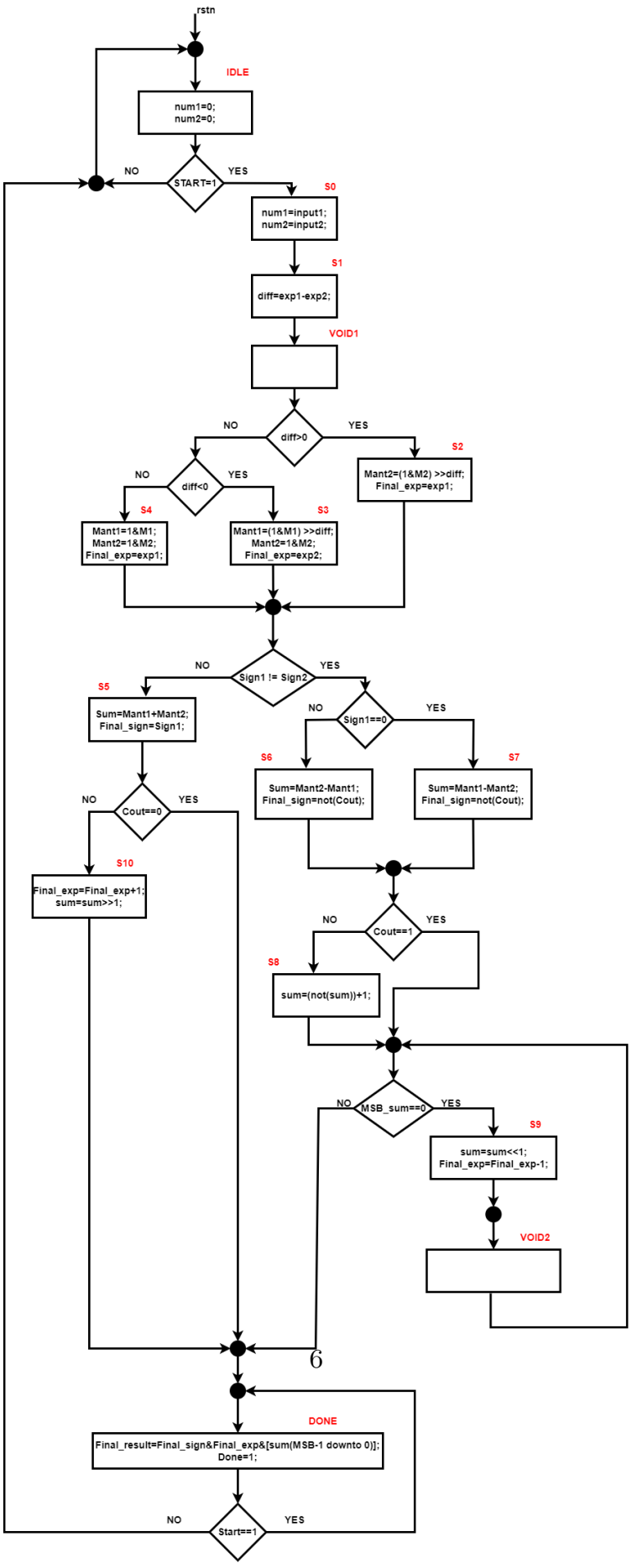

Figure 2.2: Floating point adder ASM chart

In the previous steps and ASM chart, the following cases are not included:

1. One operand or both operands are equal to **zero**. In this case, with both operands zero, the sum is zero, otherwise, it is equal to the non-zero operand.

2. One operand or both operands have values like **+infinite**, **-infinite**, **NaN**. In this case, the possible combinations of the operands with their respective sums are represented by the table 2.1

operand1	**operand2**	**sum**
$+\infty$	-	$+\infty$
-	$+\infty$	$+\infty$
$+\infty$	$+\infty$	$+\infty$
$+\infty$	$-\infty$	NaN
$-\infty$	-	$-\infty$
-	$-\infty$	$-\infty$
$-\infty$	$-\infty$	$-\infty$
$-\infty$	$+\infty$	NaN
NaN	-	NaN
-	NaN	NaN
$+\infty$	NaN	NaN
NaN	$+\infty$	NaN
$-\infty$	NaN	NaN
NaN	$-\infty$	NaN
NaN	NaN	NaN

- all the numbers except special values

Table 2.1: Behavior of a floating-point sum

3. **Overflow**. This is the case in which the sum of two positive numbers exceeds the positive limit of $3.4 \cdot 10^{38}$, or the sum of two negative numbers exceeds the

negative limit of $-3.4 \cdot 10^{38}$. The correct result is fixed to the corresponding limit.

4. **Underflow**. This is the case in which the difference of two numbers exceeds the positive limit of $1.18 \cdot 10^{-38}$ or the negative limit of $-1.18 \cdot 10^{-38}$. Also in this case the correct result is fixed to the corresponding limit.

These four cases have been studied together in the thesis work [11], and the implementation of the floating-point adder is inspired by this work, and of course it has changed.

2.3 CORDIC

The COordinate Rotation DIgital Computer (CORDIC) algorithm is an iterative algorithm, with which it is possible to compute transcendental functions such as trigonometric functions, logarithmic and exponential functions, square roots, multiplications, and divisions.
In particular, in this thesis work the architectures use it to compute sine and cosine functions.
It was initially designed to determine the exact coordinates of a vector after its rotation by an alpha angle in the Cartesian plane, as represented in figure 2.3

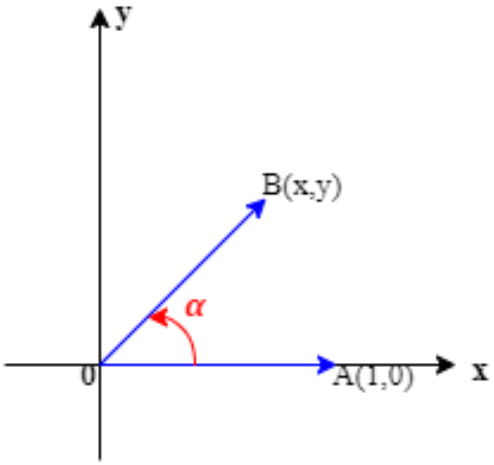

Figure 2.3: Rotation of a vector in the Cartesian plane

In mathematics there are equations that can calculate these coordinates, and they are known as **Givens Rotations**, described by the system of equations 2.3, in which $x^{(i)}$ and $y^{(i)}$ are the coordinates before the rotation, $\alpha^{(i)}$ is the angle of rotation, $x^{(i+1)}$ and $y^{(i+1)}$ are the coordinates got after the rotation.

$$\begin{cases} x^{(i+1)} = x^{(i)} \cdot cos(\alpha^{(i)}) - y^{(i)} \cdot sin(\alpha^{(i)}) \\ y^{(i+1)} = y^{(i)} \cdot cos(\alpha^{(i)}) + x^{(i)} \cdot sin(\alpha^{(i)}) \end{cases} \tag{2.3}$$

The coordinates of the vector related to the final destination are associated with the trigonometric functions having alpha as the angle. Since the rotation of the vector from position A to position B is not so easy in a single step, this algorithm proposes to approximate this rotation in a sequence of rotations, with each of them having its own angle.
It is possible to rewrite the system of equations 2.3, making a trigonometric transformation, in the system represented in 2.4

$$\begin{cases} x^{(i+1)} = \frac{x^{(i)}) - y^{(i)} \cdot tan(\alpha^{(i)})}{\sqrt{1+tan^2(\alpha^{(i)})}} \\ y^{(i+1)} = \frac{y^{(i)}) - x^{(i)} \cdot tan(\alpha^{(i)})}{\sqrt{1+tan^2(\alpha^{(i)})}} \end{cases} \tag{2.4}$$

But these formulas cannot be used yet, because there is the tangent function, and the algorithm was precisely designed to calculate functions just like the tangent. Thus it is possible to make a first simplification in the equations 2.4, and it consists of removing the two denominators, getting the system of equations shown in 2.5 and known as **pseudo-rotations**

$$\begin{cases} x^{(i+1)} = x^{(i)}) - y^{(i)} \cdot tan(\alpha^{(i)}) \\ y^{(i+1)} = y^{(i)}) - x^{(i)} \cdot tan(\alpha^{(i)}) \end{cases} \tag{2.5}$$

Without these denominators, the computations related to the coordinates of the final destination will be easier, but the modulus of its vector will be larger than expected. This effect is shown in figure 2.4
In pseudo-rotations it is possible to make another simplification: it consists of selecting only those angles whose tangent can be easily calculated. In other words, this means that $tan(\alpha^{(i)})$ is equal to 2^{-i}, and therefore $\alpha^{(i)}$ can be computed as $tan^{-1}(2^{-i})$. After these simplifications, the final system of equations related to the coordinates of the final destination is described in 2.6

$$\begin{cases} x^{(i+1)} = x^{(i)}) - y^{(i)} \cdot 2^{-i} \cdot d_i \\ y^{(i+1)} = y^{(i)}) + x^{(i)} \cdot 2^{-i} \cdot d_i \\ z^{(i+1)} = z^{(i)} - d_i \cdot \alpha^{(i)} \end{cases} \tag{2.6}$$

In this system, d_i is the direction of the rotation, which can be clockwise or counterclockwise and its values are equal to +1 or -1. A third equation is present within the system and computes $z^{(i+1)}$: this is the amount of rotation in degrees that the moving vector lacks in order to reach the final destination. The main

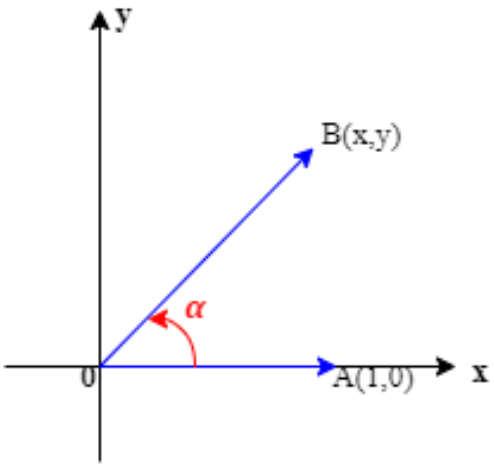

Figure 2.4: Pseudo-rotations effect

advantage of using the system of equations 2.6 is that there is no multiplication to be done because d_i changes only the signs into the equations, and the factor 2^{-i} is a simple right shift.
CORDIC operates in two modes to compute transcendental functions, known as **Rotation mode** and **Vectoring mode**. The Rotation mode uses the formulas described in 2.6 and it has the following features:

1. the initial position of the vector is along the x-axis, therefore $y^{(0)} = 0$

2. it stops iterations in case the vector has reached exactly the final destination or the amount of rotation required to reach it, is approximately zero

3. the direction of the rotation d_i is equal to the sign of $z^{(i)}$

A graphical representation of the vector after two iterations is shown in figure 2.5
The only thing left to do is to compensate for the error K_m, described in 2.7, generated by each iteration as a consequence of the first simplification. 2.5

$$K_m = \prod_{i=1}^{m} \frac{1}{\sqrt{1 + tan^2(\alpha^{(i)})}} = 0.6073 \tag{2.7}$$

To solve it, mathematically it is enough to multiply the coordinates x and y got in the last iteration by K_m.
Also the Vectoring mode uses the formulas described in 2.6 and it has these features:

1. the initial amount of rotation is zero, therefore $z^{(0)} = 0$

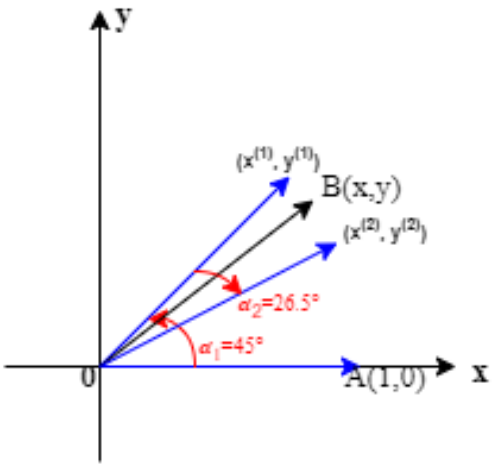

Figure 2.5: Vector rotation in Rotation mode

2. it stops iterations in case the value of the y-coordinate is exactly or approximately equal to zero.

3. the direction of the rotation d_i is equal to the opposite value of the sign of the product between $x^{(i)}$ and $y^{(i)}$

Both modes arrive at the same results: the calculated x-coordinate corresponds to the cosine function, instead, the calculated y-coordinate corresponds to the sine function.
In general, the following system of equations is used to calculate different types of functions

$$\begin{cases} x^{(i+1)} = x^{(i)}) - \mu \cdot y^{(i)} \cdot 2^{-i} \cdot d_i \\ y^{(i+1)} = y^{(i)}) + x^{(i)} \cdot 2^{-i} \cdot d_i \\ z^{(i+1)} = z^{(i)} - d_i \cdot \alpha^{(i)} \end{cases} \tag{2.8}$$

with

$$\mu \in \{-1{,}0{,}1\} \tag{2.9}$$

Considering the error factors, generated by the simplifications made to derive the system in 2.8 and shown in 2.10,

$$\begin{cases} K = 1.6468 \\ \frac{1}{K} = 0.6073 \\ K' = 0.8282 \\ \frac{1}{K'} = 1.2075 \end{cases} \tag{2.10}$$

the table 2.2 summarizes the settings to be followed to calculate the main functions correctly.

Functions	**Mode**	μ	$\alpha^{(i)}$	**Initializations**
$\cos(\beta)$, $\sin(\beta)$	Rotation	1	$tan^{-1}(2^{-i})$	x = $\frac{1}{K}$, y = 0, z = β
Multiplication	Rotation	0	2^{-i}	y = 0, x = number1, z = number2
$\cosh(\beta)$, $\sinh(\beta)$	Rotation	-1	$tanh^{-1}(2^{-i})$	x = $\frac{1}{K'}$, y = 0, z = β
$tan^{-1}(\beta)$	Vectoring	1	$tan^{-1}(2^{-i})$	x=1, z=0, y = β
Division	Vectoring	0	2^{-i}	z = 0, x = number1, y = number2
$tanh^{-1}(\beta)$	Vectoring	-1	$tanh^{-1}(2^{-i})$	x=1, z=0, y = β

Table 2.2: Functions setting

Obviously, not all transcendental functions are represented in the table 2.2 because some of them can be derived from functions already present. For example, the exponential function can be computed as the sum of the hyperbolic sine and hyperbolic cosine, or the tangent function can be obtained as the ratio between the sine and cosine functions. Interestingly, in Rotation mode, approximation errors are solved by simply initializing the variables with their respective error factors.

2.4 Existing works

Some articles in the literature have been studied to understand the various methods used to compute transcendental functions and their corresponding implementations. The table 2.3 describes the main information found in the articles.

Algo	**Date**	**Dtype**	**Op**	**Area** [mm^2]	**Power** [mW]	**Latency**	**Frequency** [MHz]
CORDIC [1]	2010	Compl2	Sen Cos	-	-	400ns	-
CORDIC [2]	2018	FixedP	Sen Cos Sen^{-1} Cos^{-1} Tan^{-1}	-	-	120ns	197.6
CORDIC [8]	2012	FloatP	Sen Cos Tan^{-1}	-	-	-	-
CORDIC [7]	2019	FloatP	Sen Cos Exp Log	-	48.75 48.75 48.65 49.92	14ck 14ck 10ck 18ck	-
CORDIC [9]	2013	FloatP	Sen Cos	-	-	-	-
CORDIC [5]	2017	FixedP	Senh Cosh Arcsenh Arccosh Arctanh	-	-	120ns	195.48
CAM [4]	2015	FixedP	Sen	-	8	-	-
CORDIC [6]	2002	FixedP	-	-	-	-	-
CORDIC [3]	2021	FixedP	Sen Cos	-	-	1ck	-

Table 2.3: Articles Overview

The table 2.4 shows, for each method, the functions that can be computed, the resources expended in terms of hardware such as multipliers, adders, and shifters, and also the clock cycles spent and the throughput got.

Algorithm	**Ops**	**App**	**Ops per Cycle**			**Cycles**	**Th**
			Mul	**Sum**	**Sh**		
CORDIC	Sen Cos Tan Exp Log Senh Cosh Tanh	[1], [2], [8] , [7] [9] ,[5], [6], [3]	0	3	3	N	$\frac{1}{T_{CK}}$
CAM	Sen Cos Tan Exp Log Senh Cosh Tanh	[4]	0	0	0	1	$\frac{1}{T_{CK}}$

N is the number of iterations

Table 2.4: Algorithms Overview

The CAM-based method [4] was not used because its implementation might require large memory sizes in order to obtain more accurate results. Moreover, with this type of implementation, it is necessary to fill memory with new values each time a different function is computed. Instead, CORDIC-based implementation makes it easier the computation of different transcendental functions because it is sufficient to direct the architecture to calculate the corresponding equation shown in 2.8 and initialize the inputs correctly, as represented in the table 2.2. In the articles studied, the main architectures based on CORDIC are differentiated according to their intended goal. Typically, most of them fall into two types: the first type aims to have high frequency and small area, while the second type focuses on reducing latency. Their implementations are depicted respectively in figures 2.6 and 2.7

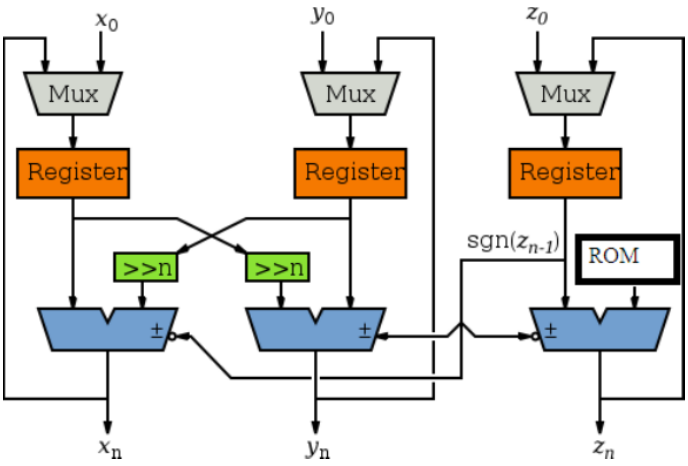

Figure 2.6: CORDIC architecture focused on area and frequency optimization [7]

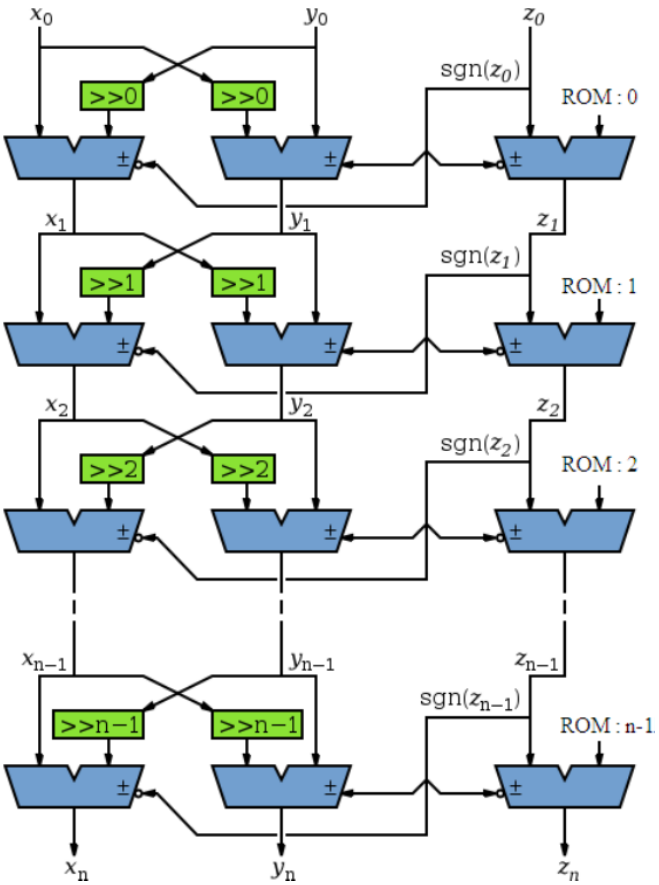

Figure 2.7: CORDIC architecture focused on latency optimization [7]

In these architectures, ROM contains $\alpha^{(i)}$ values described in the table 2.2 It is easy to see that the second architecture shown in figure 2.7 is an unrolled version of the first architecture shown in figure 2.6, and it is obtained by duplicating the first architecture a number of times equal to the established number of iterations. Since the goal of this thesis work is to implement a CORDIC architecture capable of both working at high frequency on the order of 1GHz and employing as few clock cycles as possible to compute multiple transcendental functions, both architectures were chosen as a reference from which to start. But they work well with fixed-point and two complement formats, not for floating-point. For this reason, both architectures have been adapted to the floating-point format, and all the details about this are described in the following chapter.

Chapter 3

Architectures proposal

As already anticipated, first the architecture in figure 2.6 was adapted to the floating-point format. To succeed in this, the following changes have been made:

1. ROM has been replaced by a lookup table (LUT).
2. Obviously both the inputs x_0, y_0, z_0, and the values stored in the LUT are in floating-point format.
3. Binary adders have been replaced by floating-point adders.
4. In the floating-point format, divisions for a number by a power of base two cannot be done by adopting the same method used for binary numbers, that is by shifting the number to the right by as many positions as the value of the exponent. Instead, they are done simply by subtracting the exponent of the floating-point number from the exponent of the power of two. Therefore, it is necessary to replace the right shifters with binary subtractors.

Moreover, the number of iterations performed is counted by a simple binary counter.

3.1 Floating point adder

The floating point adder implementation has been developed taking as reference the previous ASM chart represented in figure 2.2. The whole implementation of the asynchronous floating point adder is represented by the figures 3.1, 3.2, 3.3, 3.4.

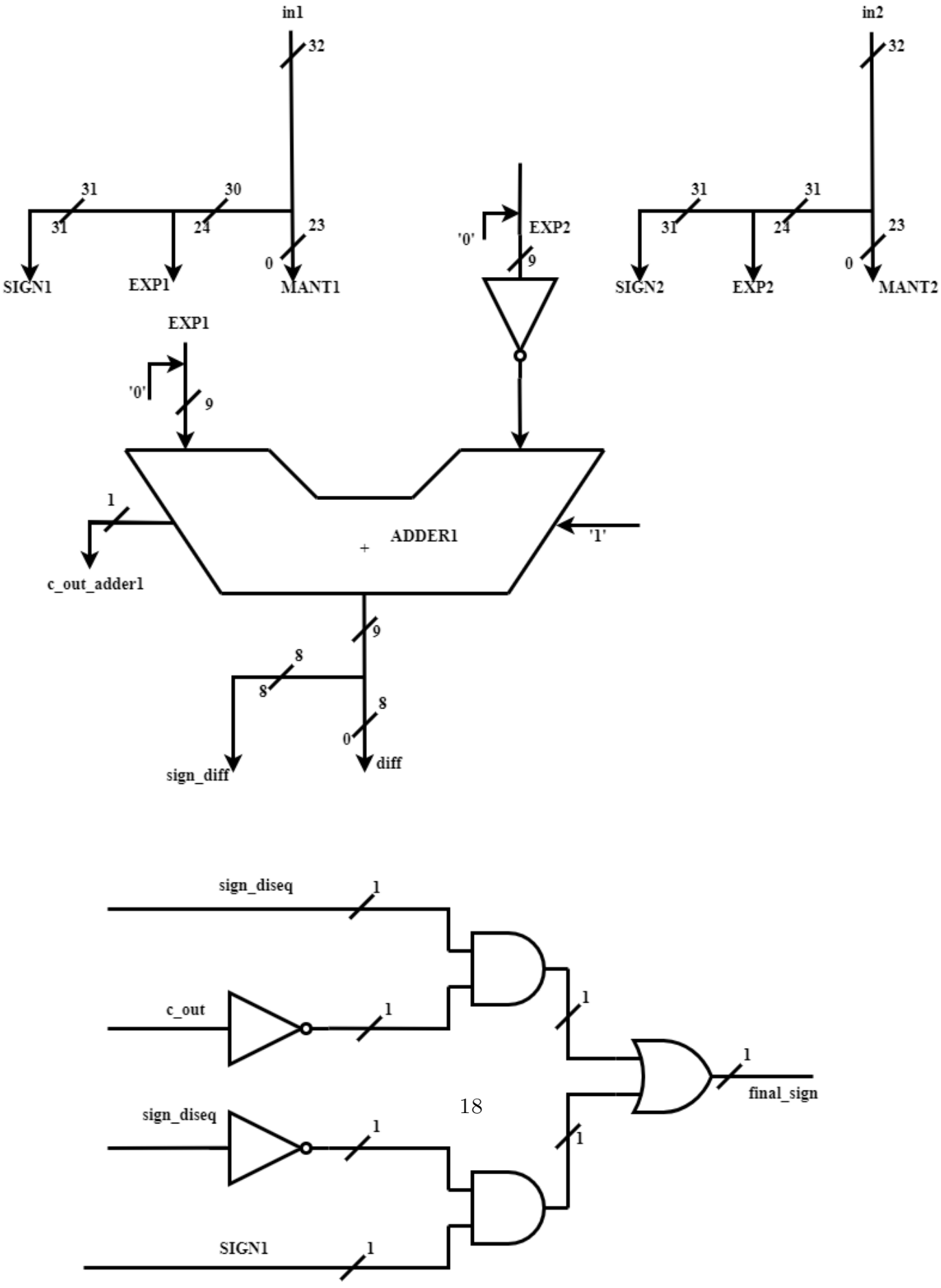

Figure 3.1: Asynchronous floating-point adder(part 1)

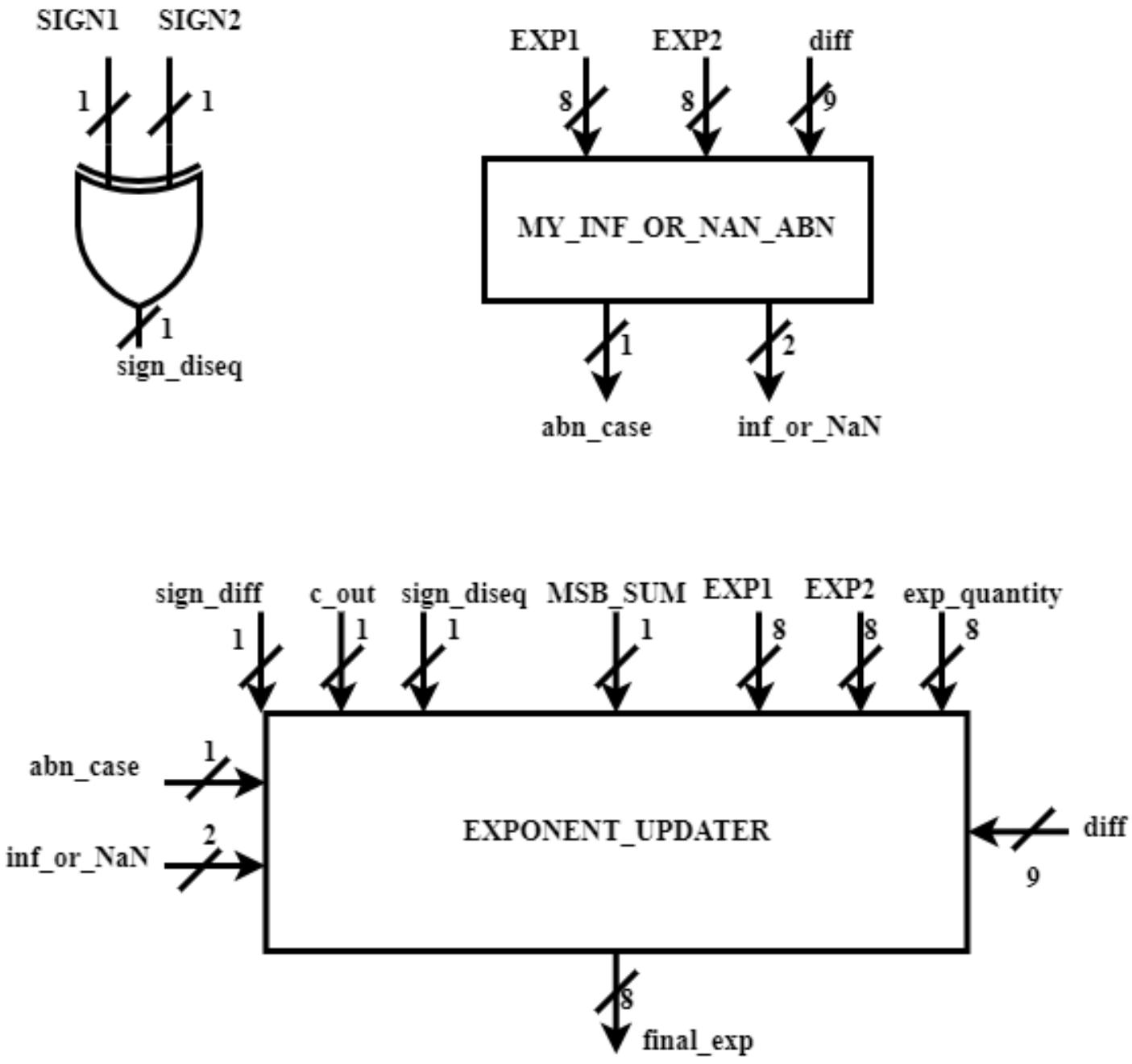

Figure 3.2: Asynchronous floating-point adder(part 2)

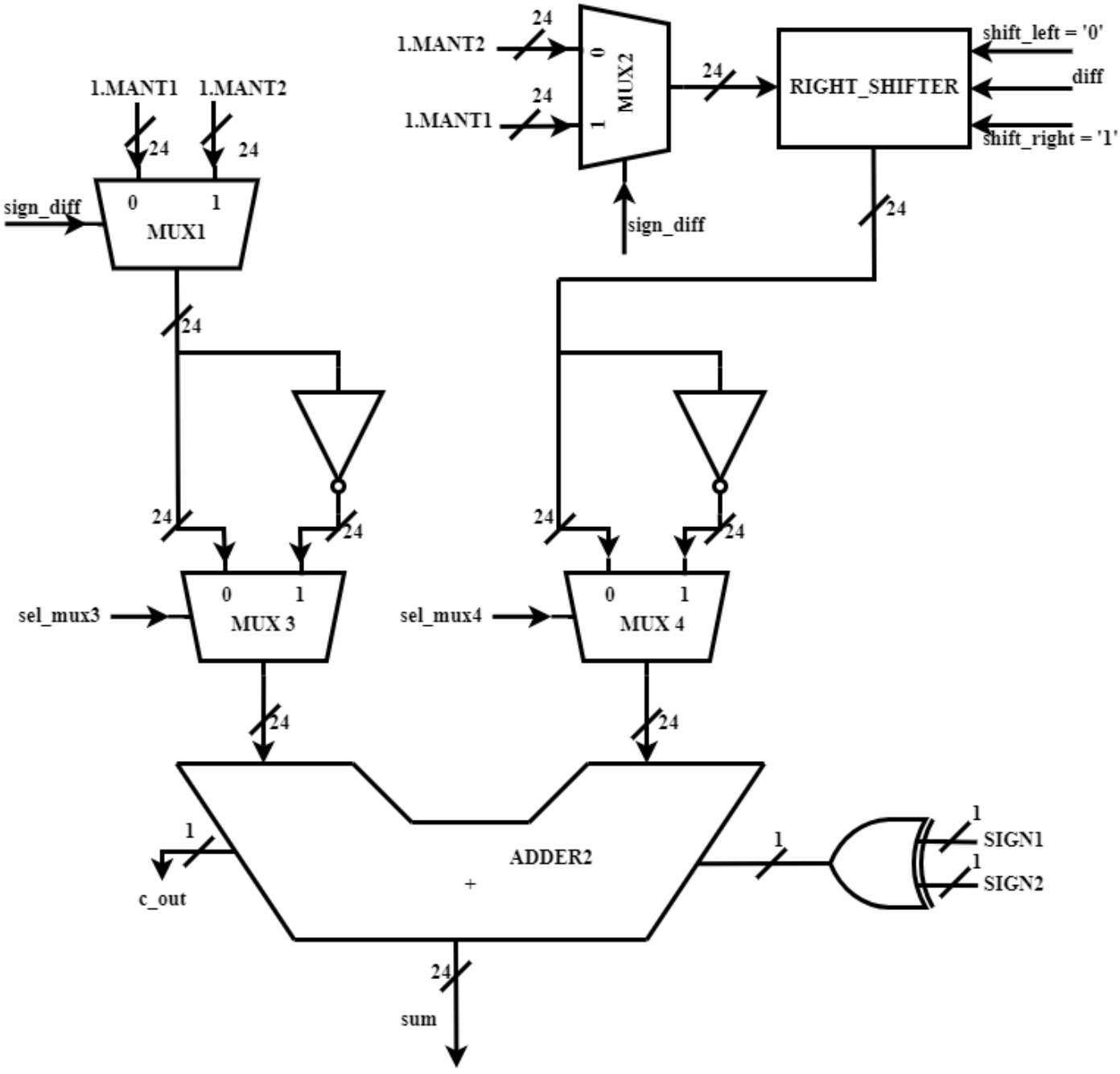

Figure 3.3: Asynchronous floating-point adder(part 3)

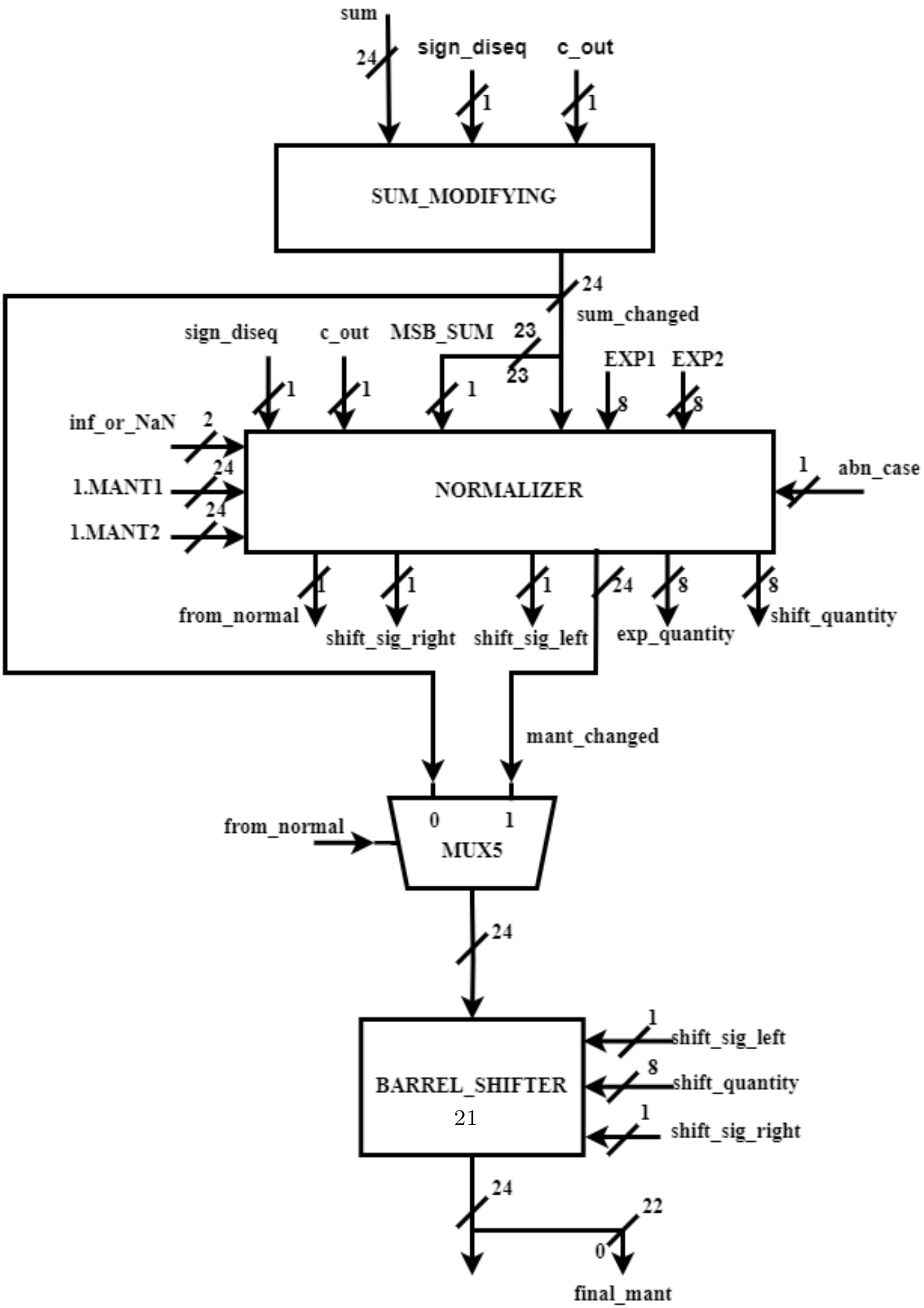

Figure 3.4: Asynchronous floating-point adder(part 4)

It is composed of the following components:

1. The **adder** named **ADDER1**, used to make the difference between the two exponents, to establish which of the two is greater or if they are equal. It is important to note that this adder has a parallelism equal to nine, and not to eight like the exponents. There is an extension of one bit and it is the zero logic because the exponents are unsigned binary numbers. In this way, both become numbers with positive signs and thus it is possible to compute correctly the difference and its sign.

2. The **multiplexers MUX1** and **MUX2**, and they have an extension of one bit. Their inputs are on 24 bits and not on 23 bits because a bit equal to one is added in the MSB position at each mantissa. Of course, only one of the two mantissa is sent to the first barrel shifter because in the worst case, only the exponent of one addend is lower than the exponent of the other addend.

3. The **RIGHT_SHIFTER**, that is a barrel shifter with the shift_right signal fixed at logical one. It used to shift to the right its input by a value equal to the absolute value of the difference of the exponents.

4. The **multiplexers MUX3** and **MUX4**, whose outputs correspond respectively to the outputs of MUX1 and the RIGHT_SHIFTER if the two addends have the same sign. Otherwise, one of them will pass the mantissa belonging to the addend with the negative sign with all its bits replaced by those of opposite logical value. Their selection signals have been implemented simply using a process and the cases treated are described in detail in the following table 3.1

sign_diff	**sign_diseq**	**sel_mux3**	**sel_mux4**
0	0	0	0
0	1	SIGN1	SIGN2
1	0	0	0
1	1	SIGN2	SIGN1

Table 3.1: Behavior of the selection signals sel_mux3 and sel_mux4

5. A second **adder** called **ADDER2**, with which it is possible to make two operations: the sum of the two mantissa, if the addends have the same sign. In this case, the carry-in of the adder is set equal to zero; the subtraction of the two mantissa. Of course, the carry-in of the adder is set equal to one.

6. The **component** called **SUM_MODIFYING**. It accepts the output signal sum of the adder2, its c_out, and the signal sign_diseq.
   In the case in which sign_diseq is equal to one and c_out is zero, its output will be equal to the value of the sum generated by adder2 but with opposite sign. As previously explained, this means that each bit of the sum is replaced with its opposite logical value, and the number is added to one. Otherwise, its output will be equal to the sum without any changes.

7. The **component** called **MY_INF_OR_NAN_ABN**. It is possible that each input of the floating-point adder can have special values as $+\infty$, or $-\infty$ or NaN. In the case in which both addends have special values like NaN or $+\infty$ or $-\infty$, the output signal abn_case is put equal to logical one .
   Instead, if only the addend1 has a special value, the binary value of the output signal inf_or_NaN is put equal to 01, otherwise if only the addend2 has a special value, the binary value of the output signal inf_or_NaN is put equal to 10.
   In order to make its behavior clearer, the table 3.2 summarizes all the information already mentioned about this component.

number1	number2	result	abn_case	inf_or_NaN
$+\infty$	-	$+\infty$	0	01
-	$+\infty$	$+\infty$	0	10
$+\infty$	$+\infty$	$+\infty$	1	00
$+\infty$	$-\infty$	NaN	1	00
$-\infty$	-	$-\infty$	0	01
-	$-\infty$	$-\infty$	0	10
$-\infty$	$-\infty$	$-\infty$	1	00
$-\infty$	$+\infty$	NaN	1	00
NaN	-	NaN	0	01
-	NaN	NaN	0	10
$+\infty$	NaN	NaN	1	00
NaN	$+\infty$	NaN	1	00
$-\infty$	NaN	NaN	1	00
NaN	$-\infty$	NaN	1	00
NaN	NaN	NaN	1	00
-	-	-	0	00

- all the numbers except special values

Table 3.2: Behaviour of the component INF_OR_NAN_ABN

8. The **component** called **EXPONENT_UPDATER**. This is a fundamental unit in the floating point adder because it determines the right value of the final exponent in all the possible cases. To implement this unit, the ASM chart represented in figure 2.2 has been taken as a reference, and all its cases in which the final exponent changes have been identified. To these, the four cases previously explained in the section 2.2 have been added. The whole block has been implemented in a behavioral way, and it corresponds to a sequence of cases, reported below:

   1) With the signal abn_case equal to one, the possible results of the sum

can have values such as $+\infty$, $-\infty$, NaN. These three values have all bits of the exponent equal to one, and therefore the final exponent has the binary value 11111111.

2) If the two addends have different signs the difference between their exponents is equal to zero and the signal exp_quantity received from the NORMALIZER has an integer value of 24, this is the case of a sum of two addends with opposite signs. Therefore the binary value of the final exponent is 00000000.

3) With the signal inf_or_NaN equal to 01 or 10, just as in the first case, the possible results of the sum are $+\infty$, $-\infty$, NaN. Therefore, the final exponent is equal to 11111111.

4) This is the case of overflow, identified by the c_out of ADDER2 equal to logical one, one of the two exponents is equal to 11111110 and the two addends have the same sign. The final exponent is equal to 11111110.

5) This is the case of underflow, identified by the c_out of ADDER2 equal to logical one, both exponents are equal to 00000001 and the two addends have different signs. The final exponent is equal to 00000001.

The remaining cases have been implemented simply by translating the ASM chart conditions into the respective logic signals, which corresponds of course, to various values that the final exponent can take on. All these cases are summarized in the table 3.3

sign_diff	**sign_diseq**	**c_out**	**MSB_SUM**	**final exponent**
0	0	0	-	exp1
0	1	-	1	exp1
1	0	0	-	exp2
1	1	-	1	exp2
0	0	1	-	exp1 + 1
1	0	1	-	exp2 + 1
0	1	-	0	exp1 - exp_quantity
1	1	-	0	exp2 - exp_quantity

- don't care

Table 3.3: Values of the final exponent in the remaining cases.

9. The **component** called **NORMALIZER**. It is another fundamental component, and with its work, it is possible to compute the final mantissa correctly and contribute to the calculation of the final exponent by cooperating with the component EXPONENT_UPDATER via the exp_quantity signal.Its outputs are: the signals shift_sig_right and shift_sig_left, used to control the direction of the shifting operation of the component BARREL_SHIFTER, which will shift by an amount indicated by the other output signal shift_quantity; the signal exp_quantity, sent to the component EXPONENT_UPDATER, and corresponds to the quantity that must be subtracted in the computation of the final exponent because this is the same quantity that the NORMALIZER calculated as the left shift number to correct the mantissa; the signal from_normal, that is the selection signal of MUX5. With this signal, the NORMALIZER can decide to continue the computation of the final mantissa choosing between the output of the SUM_MODIFYING component and the mantissa mant_changed generated by itself; the signal mant_changed, that is the correct mantissa related to special cases, such as the presence of overflow or underflow, or one of the two addends has $+\infty$ or $-\infty$ as values, or both addends have opposite signs and the same modulus equal to ∞ or NaN. In detail, all cases handled by the NORMALIZER are described below:

   1) With the signals abn_case and sign_diseq equal to the logical one, and the sum generated by ADDER2 equal to zero, this is the case in which both addends have opposite signs and the same modulus equal to ∞ or NaN. The result of this sum is NaN, and therefore the mantissa mant_changed has all its bits equal to the logical one and from_normal is put equal to the logical one.
   2) With inf_or_NaN equal to 01, the result of the floating-point sum has the same value as the first addend. Therefore mant_changed is composed of a logical one in the MSB position followed by the same bits present in the mantissa of the first addend. Similarly, with inf_or_NaN equal to 10, mant_changed is composed of logical one in MSB position followed by the same bits present in the mantissa of the second addend. In both cases, from_normal is put equal to the logical one.
   3) The two cases of overflow and underflow are identified by the same conditions described previously for the EXPONENT_UPDATER component. In the first case, all the bits of mant_changed are equal to the logical one, in the second case they are equal to the logical zero. Also in these two cases, from_normal has a logical one as a value.
   4) If the two addends have the same signs and the c_out of the ADDER2 is equal to the logical one, as described in the previous ASM chart, this means that the sum_changed must be shifted to the right by one position.

Therefore shift_quantity has an integer value equal to one, and the signal shift_sig_right has logic one as value.

5) With signal_diseq equal to the logical one and MSB_SUM equal to the logical zero, a check is performed on sum_changed, which should have a logical one as MSB. The NORMALIZER calculates the position of the first logical one starting at the MSB position in the direction of the LSB. It counts from zero and increases by one unit each time, not finding the logical one, it moves to the next position. For example, if the logical one is at bit 21, the NORMALIZER will match the number two as the position. This number is transmitted to the EXPONENT_UPDATER via the exp_quantity signal and to the BARREL_SHIFTER via the shift_quantity signal along with the shift_sig_left signal put equal to the logical one, so that sum_changed can be shifted to the left and the exponent can be decremented, thus obtaining the correct mantissa and exponent.

One consideration need to be made regarding the implementation of the NORMALIZER: initially, this component was implemented in a behavioral way as done for the EXPONENT_UPDATER, that is, describing a sequence of cases, and for each case, the corresponding outputs were well defined. But in the next step of analyzing the frequency of the floating-point adder, the NORMALIZER introduced a major delay in the critical path, severely limiting the working frequency of the adder. Thus the NORMALIZER was implemented with a structural architecture, obtained by translating all the cases explained in detail previously into logic gates, and multiplexers, and also using a new component named ONE_DETECTOR_32BIT. This component accepts the 24-bit sum_changed signal and its output is on 8 bits, corresponding to the binary value of the position of the first logical one. This output is computed in a behavioral way using a series of when to check which bit of the input is at the logical one. At each of these when's corresponds to a certain value of its output. The following figures show the architectures implemented for each output of the NORMALIZER.

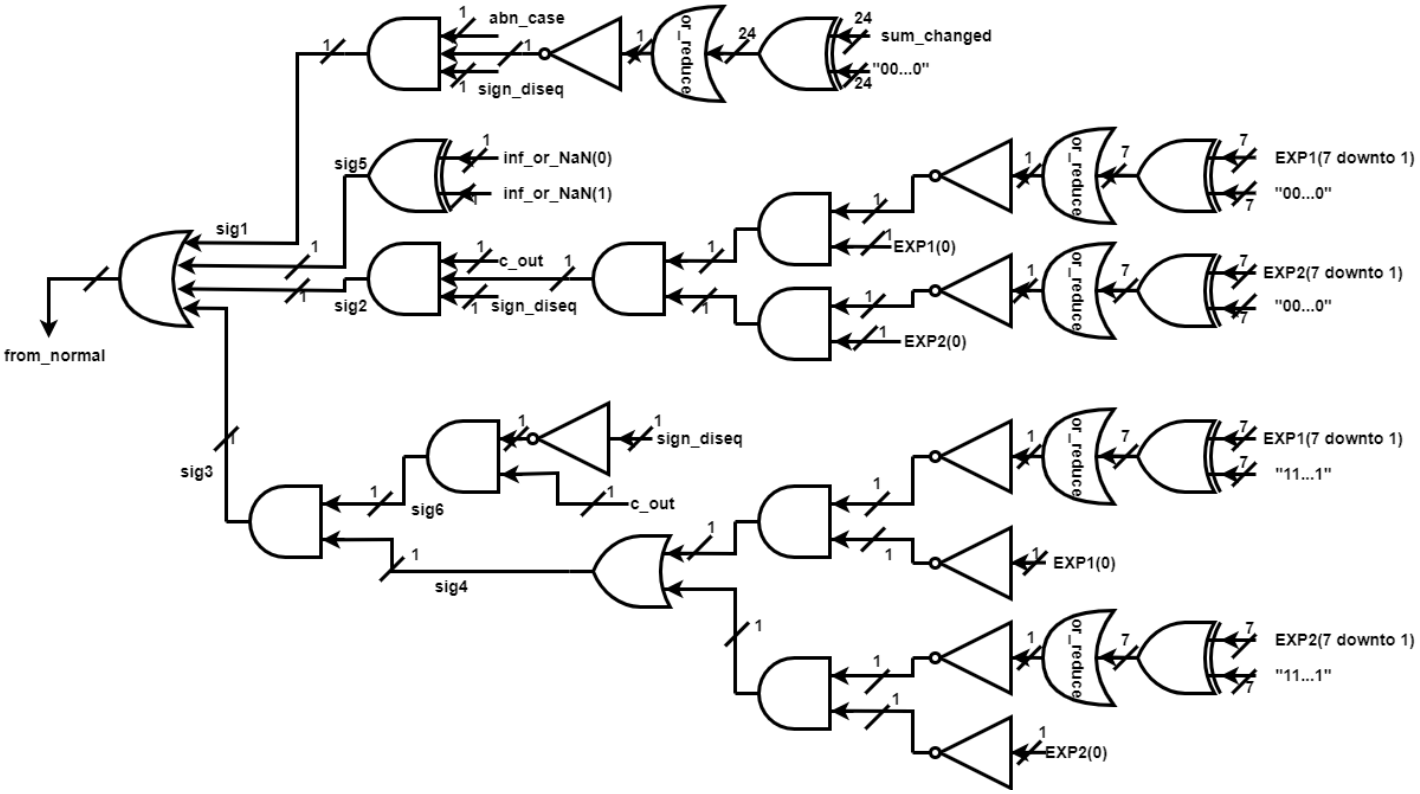

Figure 3.5: Implementation of from_normal output

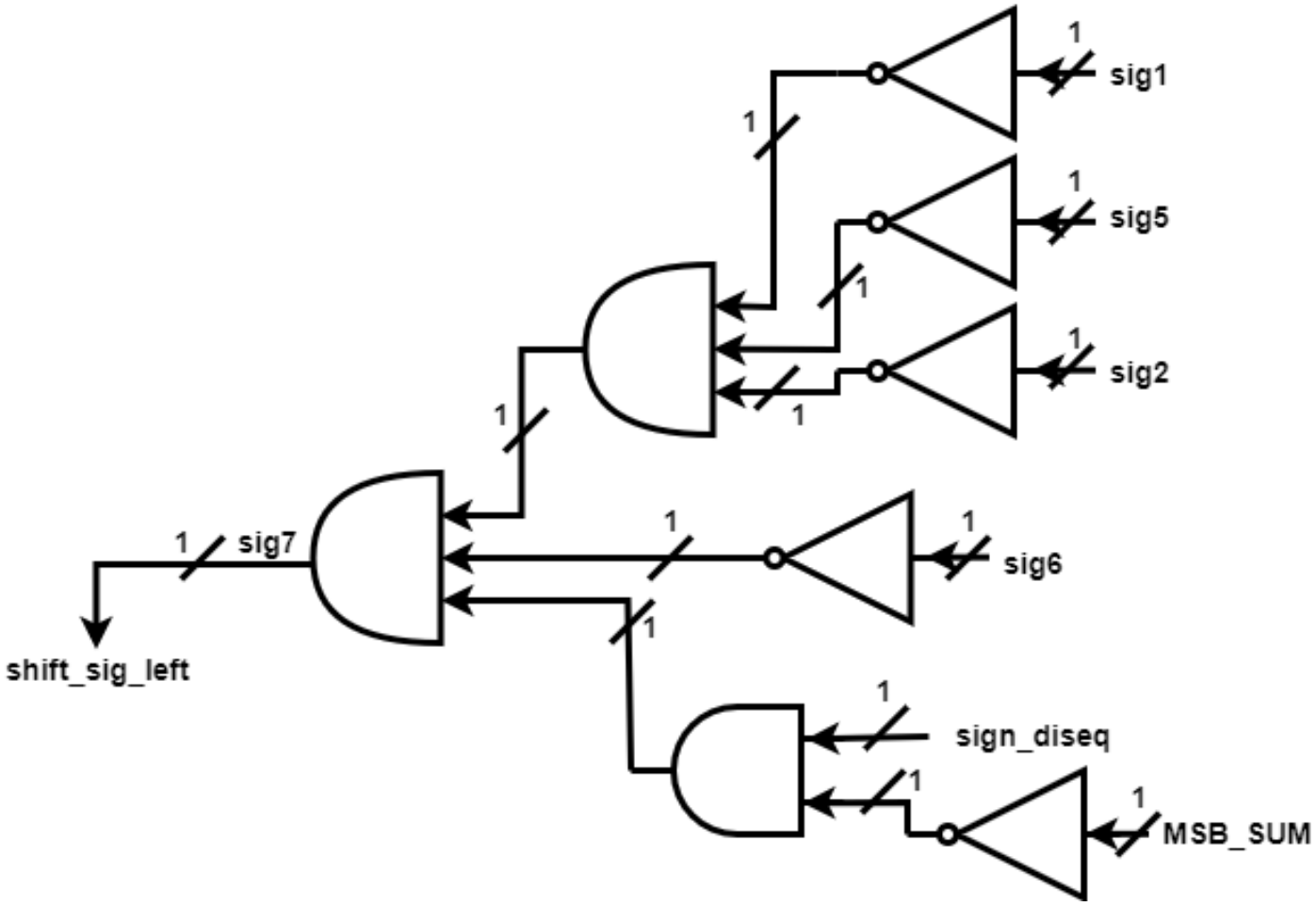

Figure 3.6: Implementation of shift_sig_left output

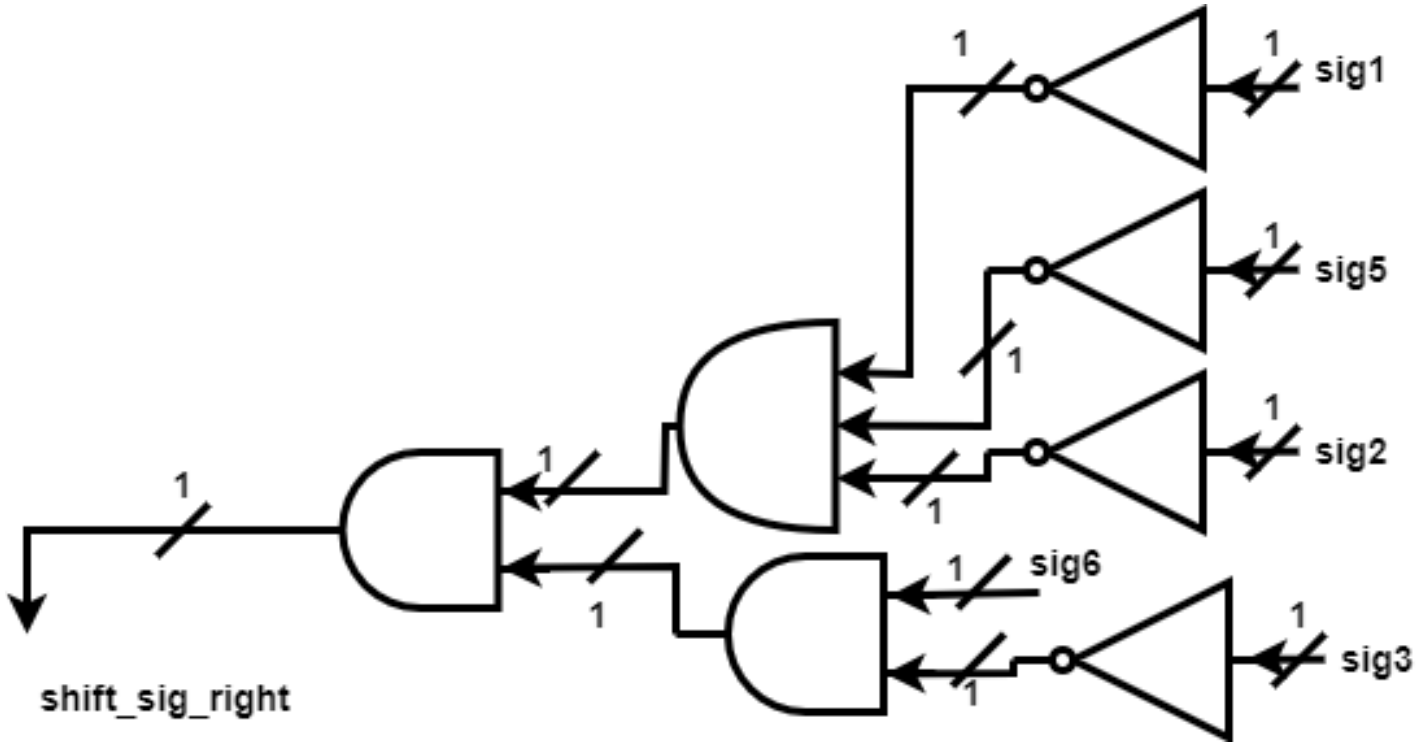

Figure 3.7: Implementation of shift_sig_right output

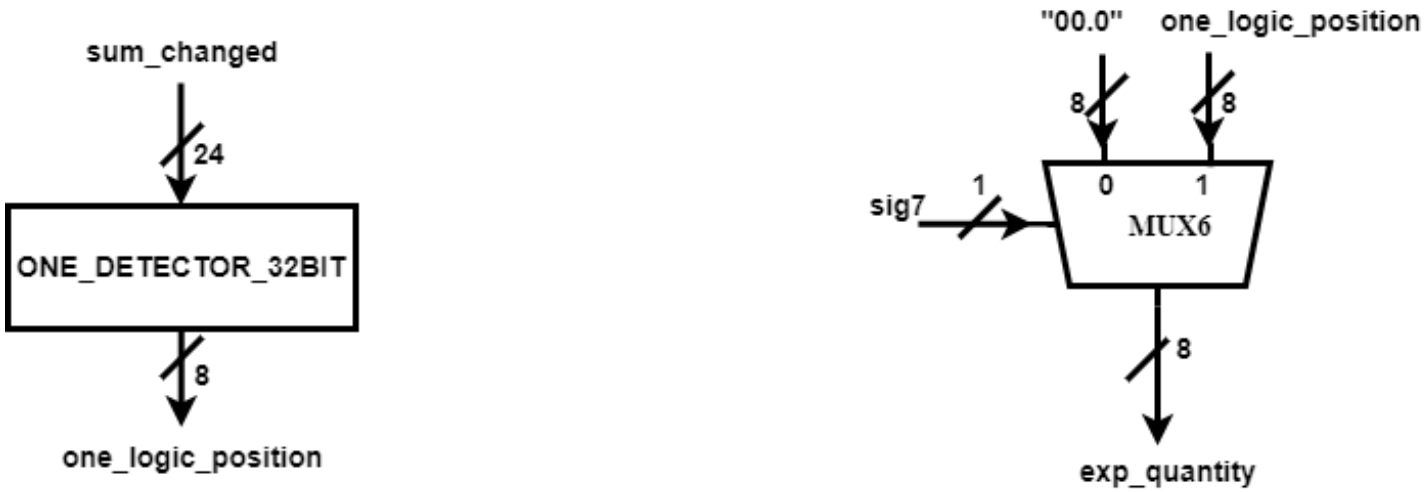

Figure 3.8: Implementation of exp_quantity output

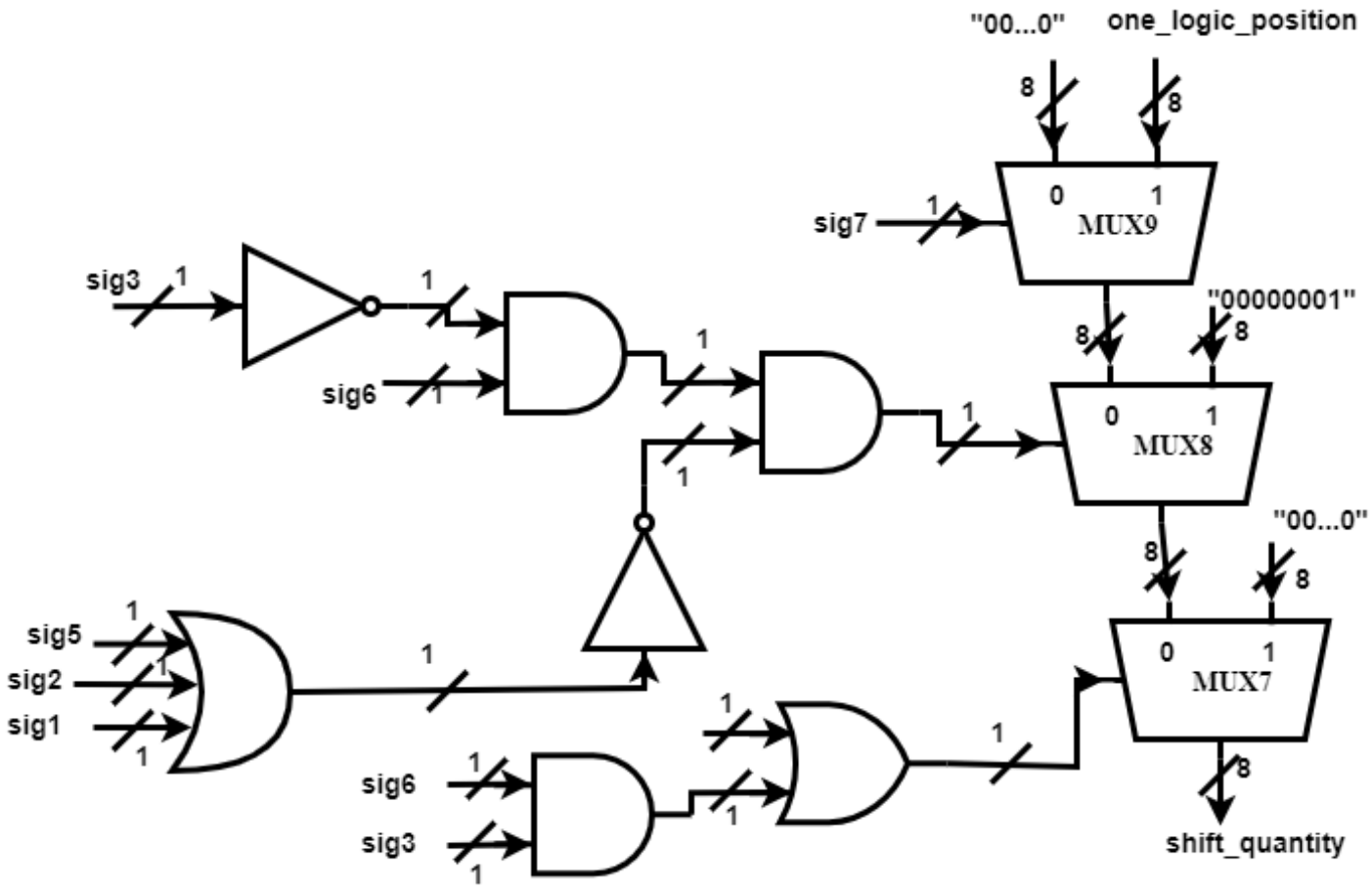

Figure 3.9: Implementation of shift_quantity output

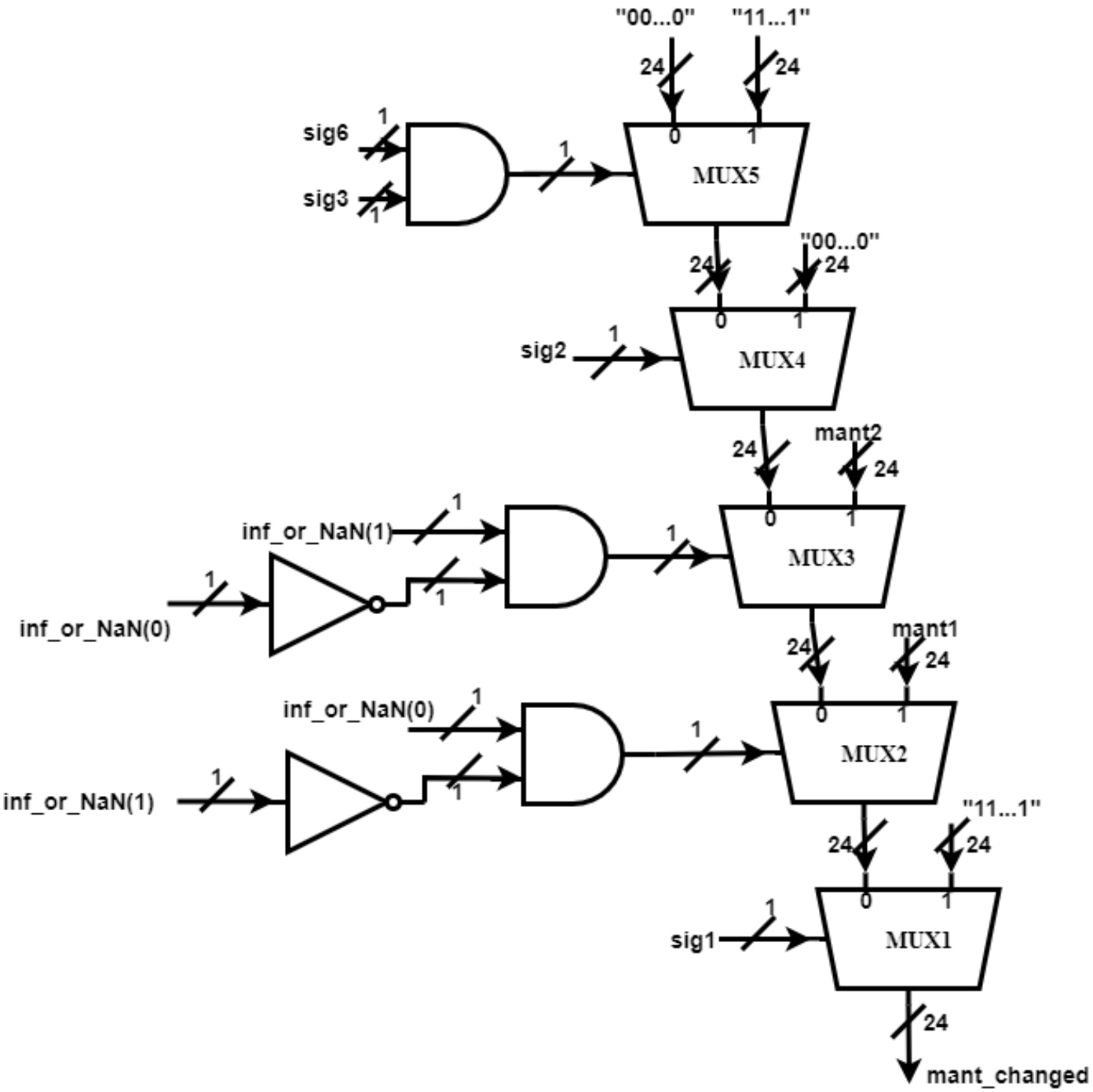

Figure 3.10: Implementation of mant_changed output

10. The **multiplexer MUX5**, whose selection signal is controlled by the NORMALIZER. Its output can be equal to the sum_changed signal or mant_changed signal.
11. the second **BARREL_SHIFTER**, used to correct the mantissa. Both its shift direction control signals and the amount to be shifted are sent by the NORMALIZER. All bits of its output signal, except the MSB, correspond to the final mantissa.
12. The logic function implemented to calculate the final sign has been derived simply by identifying the ASM chart conditions for which the final sign has certain values.

Therefore, the final result of the floating-point adder is given by the concatenation of the sign, exponent, and mantissa computed by the described architecture.

3.2 Testing floating-point adder using UVM

To be sure that the implemented floating-point adder works for any value of the inputs, it was tested by UVM testbench. Its structure is represented in figure 3.11 with a block diagram.

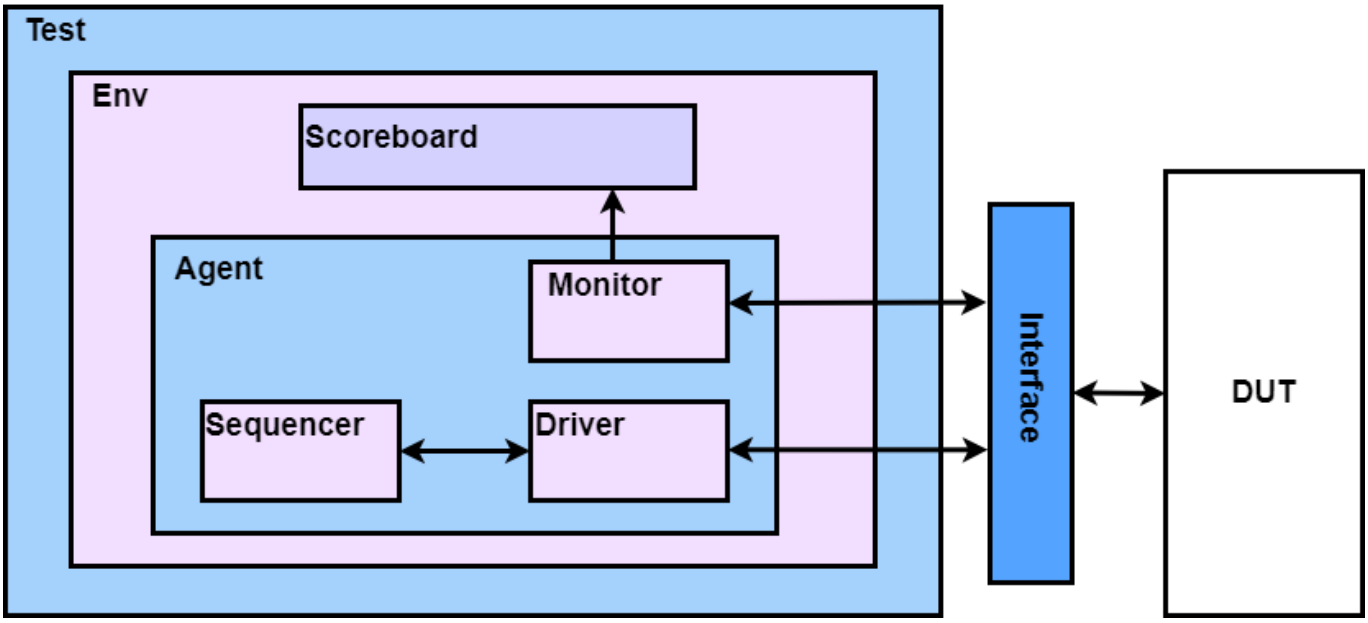

Figure 3.11: UVM testbench

This particular testbench uses an object-oriented approach, in which every component is a class. Its general behavior is as follows: at the output of the Sequencer,

there is a flow of random stimuli. These stimuli are converted by the driver into transactions, which are translated into pin signals by the interface. Subsequently, the interface translates the pin signals generated by the DUT into transactions, which arrive at the monitor input. The latter performs the inverse function of the driver and the translated values arrive at the Scoreboard input, which groups the prediction task and the evaluation task. The first task is performed by its internal component called predictor, which takes the same input transactions that are sent to the DUT and produces the expected output transactions. Instead, the evaluation task is performed by its internal component called comparator, which compares the expected and actual outputs. If both values are equal to each other, a match message is produced, otherwise, a mismatch message is generated, as shown in figure 3.12

```
# adder: input A = 01010001001000111010011010101101, input B = 00011000001101111100011011110001, output OUT = 01010001001000111010011010101101
# adder: input A = 4.3929752e+10, input B = 2.3752635e-24, output OUT = 4.3929752e+10
# refmod: input A = 01010001001000111010011010101101, input B = 00011000001101111100011011110001, output OUT = 01010001001000111010011010101101
# refmod: input A = 4.3929752e+10, input B = 2.3752635e-24, output OUT = 4.3929752e+10
# UVM_INFO @ 1125: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A = 10100100100100000111000010110001, input B = 00111110110101000010110010011011, output OUT = 00111110110101000010110010011011
# adder: input A = -6.2640952e-17, input B = 4.1440281e-01, output OUT = 4.1440281e-01
# refmod: input A = 10100100100100000111000010110001, input B = 00111110110101000010110010011011, output OUT = 00111110110101000010110010011011
# refmod: input A = -6.2640952e-17, input B = 4.1440281e-01, output OUT = 4.1440281e-01
# UVM_INFO @ 1155: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A = 10010011110010010101011001100, input B = 00011001001111111110110001001010, output OUT = 00011001001111111101001100000000
# adder: input A = -5.1077557e-27, input B = 9.9221868e-24, output OUT = 9.9170797e-24
# refmod: input A = 10010011110010010101011001100, input B = 00011001001111111110110001001010, output OUT = 00011001001111111101001011111111
# refmod: input A = -5.1077557e-27, input B = 9.9221868e-24, output OUT = 9.9170789e-24
# UVM_INFO @ 1185: reporter [MISCMP] Miscompare for exp.data: lhs = 'h193fd2ff : rhs = 'h193fd300
# UVM_INFO @ 1185: reporter [MISCMP] 1 Miscompare(s) for object tr@825 vs. exp@617
# UVM_WARNING @ 1185: uvm_test_top.env_h.comp [Comparator Mismatch]
```

Figure 3.12: Match and mismatch messages generated during UVM testbench

It is necessary to make one consideration about this testbench: most of the time, the expected and actual outputs are equal to each other, but in some cases, such as the one shown in figure 3.12, there are mismatches because the actual outputs have different digits than expected starting from the seventh decimal digit. The reason is related to the precision of the floating-point format, that is between 6 and 7 digits. Therefore, mismatches of this kind were considered as matches.

3.3 CORDIC architecture

After designing the floating-point adder, the CORDIC architecture has been implemented, represented in figure 3.13
As already anticipated, the number of iterations chosen is five, and the outputs of the X and Y registers correspond respectively to the cosine and sine functions. The ASM chart of this architecture, from which the control unit can be derived immediately, is shown in figure 3.14

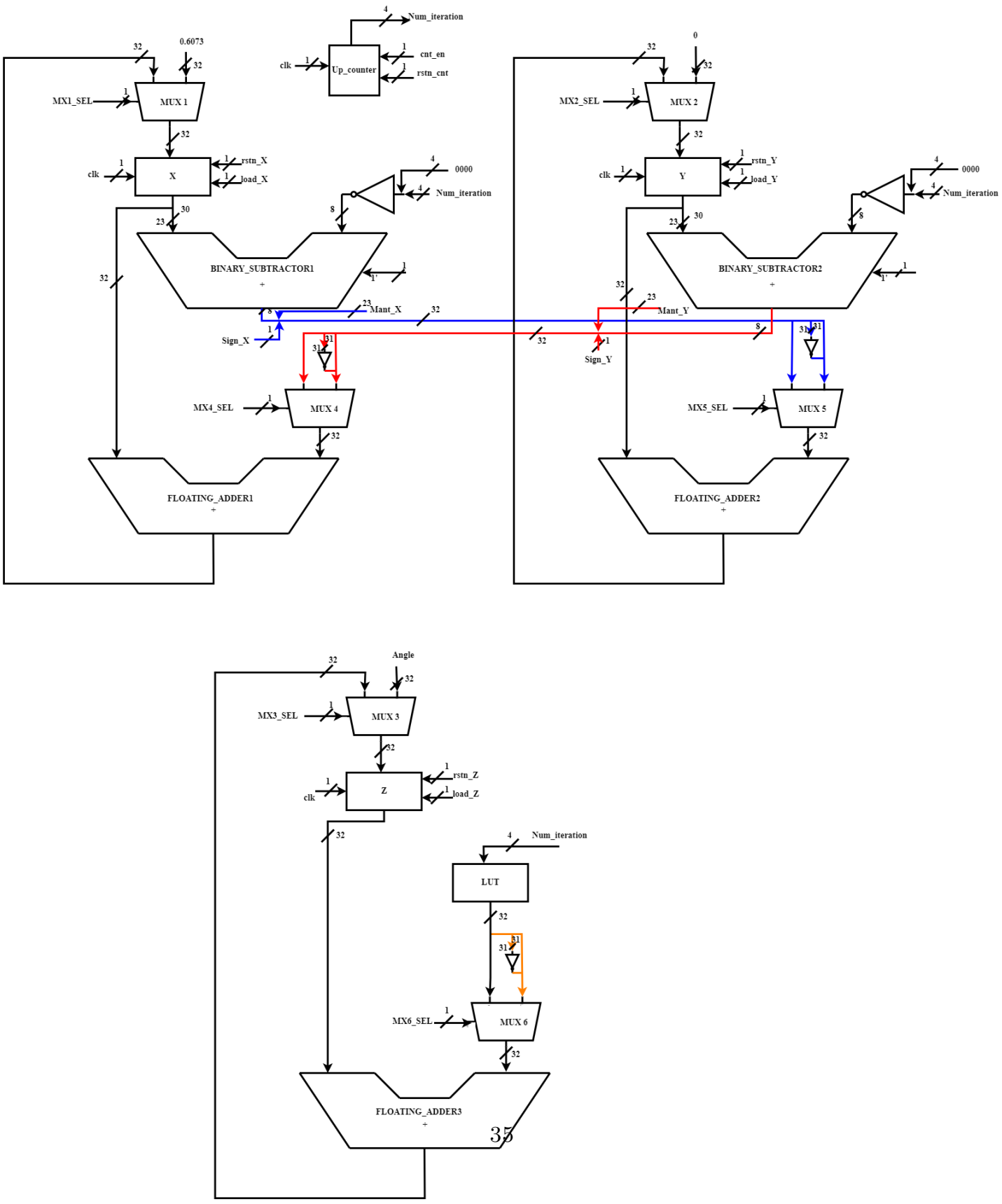

Figure 3.13: CORDIC architecture

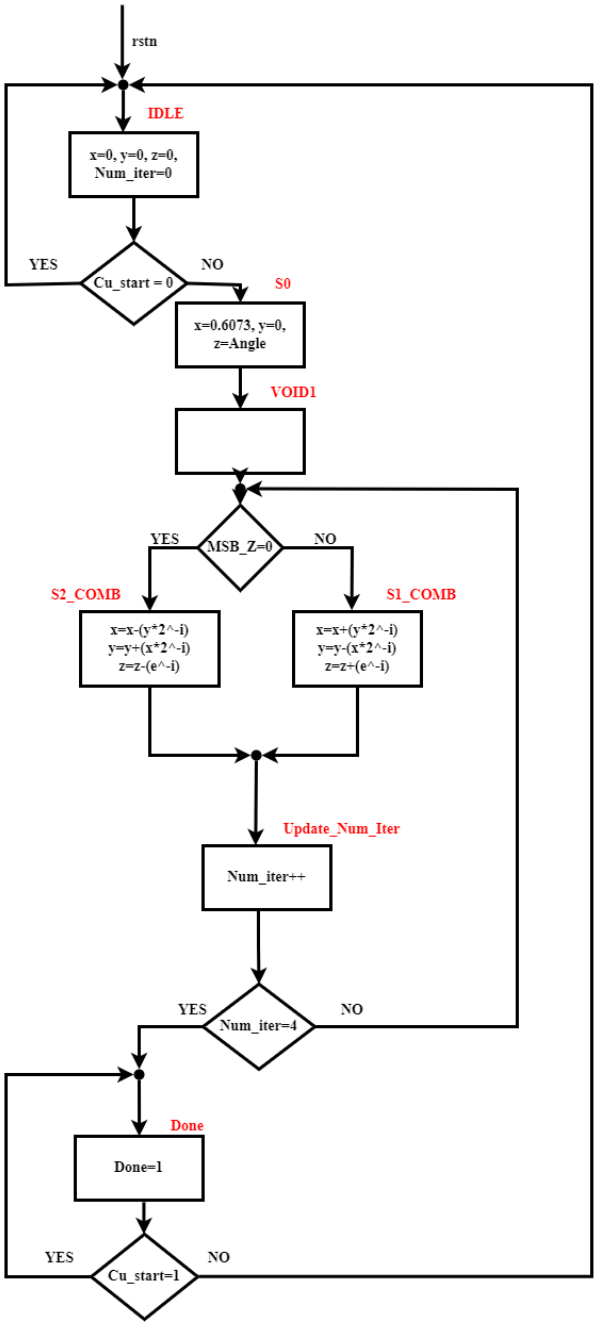

Figure 3.14: ASM chart CORDIC architecture that uses asynchronous floating-point adders

The only disadvantage of using this architecture is related to its working frequency. This is equal to 251.26MHz, so rather far from the frequency of the order of 1GHz that, instead, is desired. The interesting aspect of its critical path is that precisely the floating-point adder introduces most of the delay. Thus, pipeline stages were gradually introduced within the floating-point adder to increase the CORDIC architecture's working frequency, until a frequency close to 1GHz was achieved.

3.4 Pipelined 32-bit floating-point adders

For each pipeline stage added, special emphasis was placed on the frequency of both the modified floating-point adder and the CORDIC architecture that integrates it, so as to identify the sufficient number of pipeline stages. A total of four pipeline stages were added to the floating-point adder thus generating four versions of the floating-point adder and the CORDIC architecture. The changes made regarding the parts that constitute the floating-point adder are represented from figure 3.15 to figure 3.27 . The colored lines present within the figures are registers. Obviously, a different control unit is needed than previously used, since this time the floating-point adders are pipelined. For this reason, a new ASM chart, obtained by modifying the previous one shown in figure 3.14, was designed. It is represented in figure 3.28 and it is suitable for both CORDIC architectures that use asynchronous floating-point adders and those that use pipelined floating-point adders.

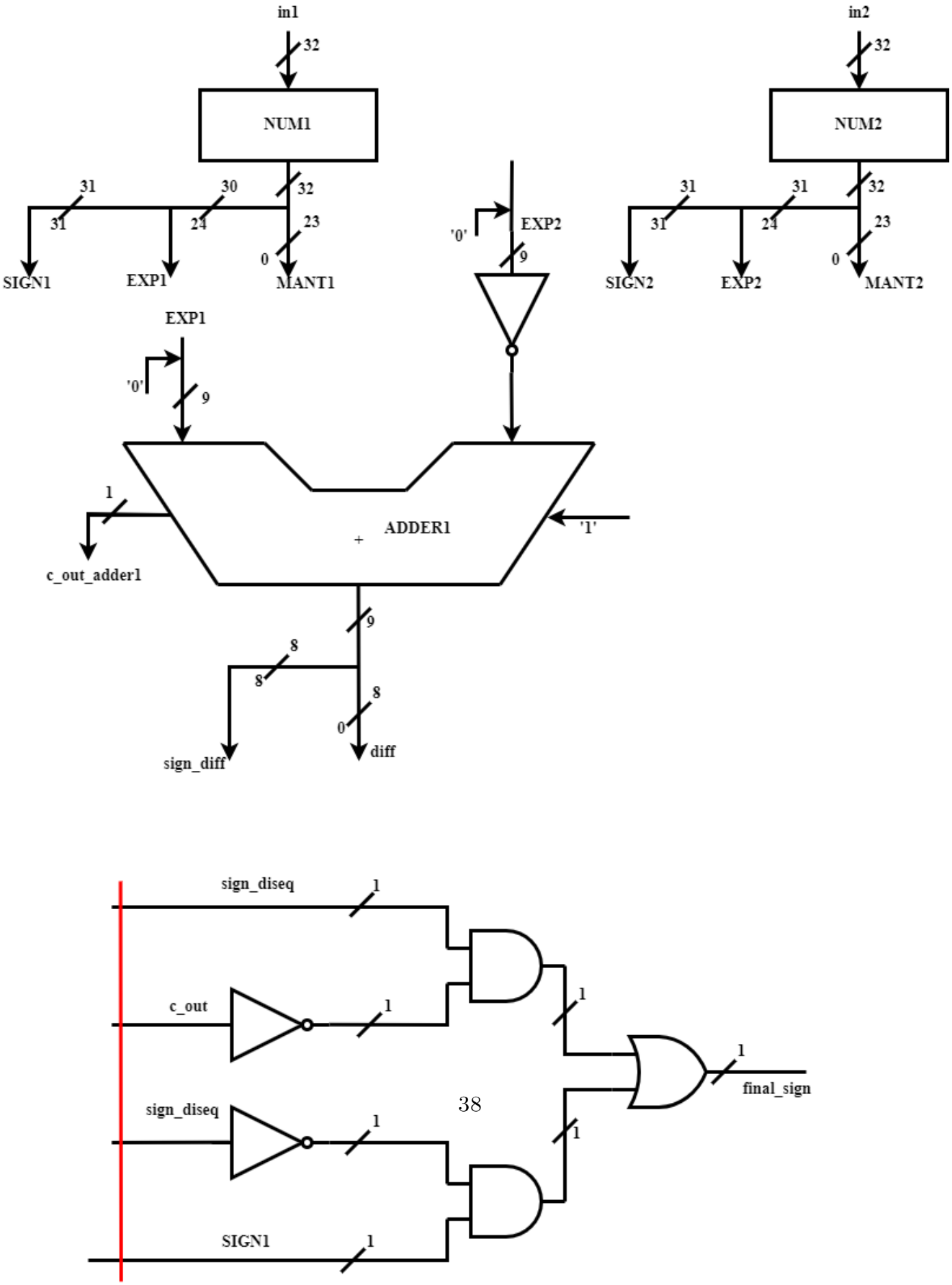

Figure 3.15: Floating-point adder with one pipeline stage(part 1)

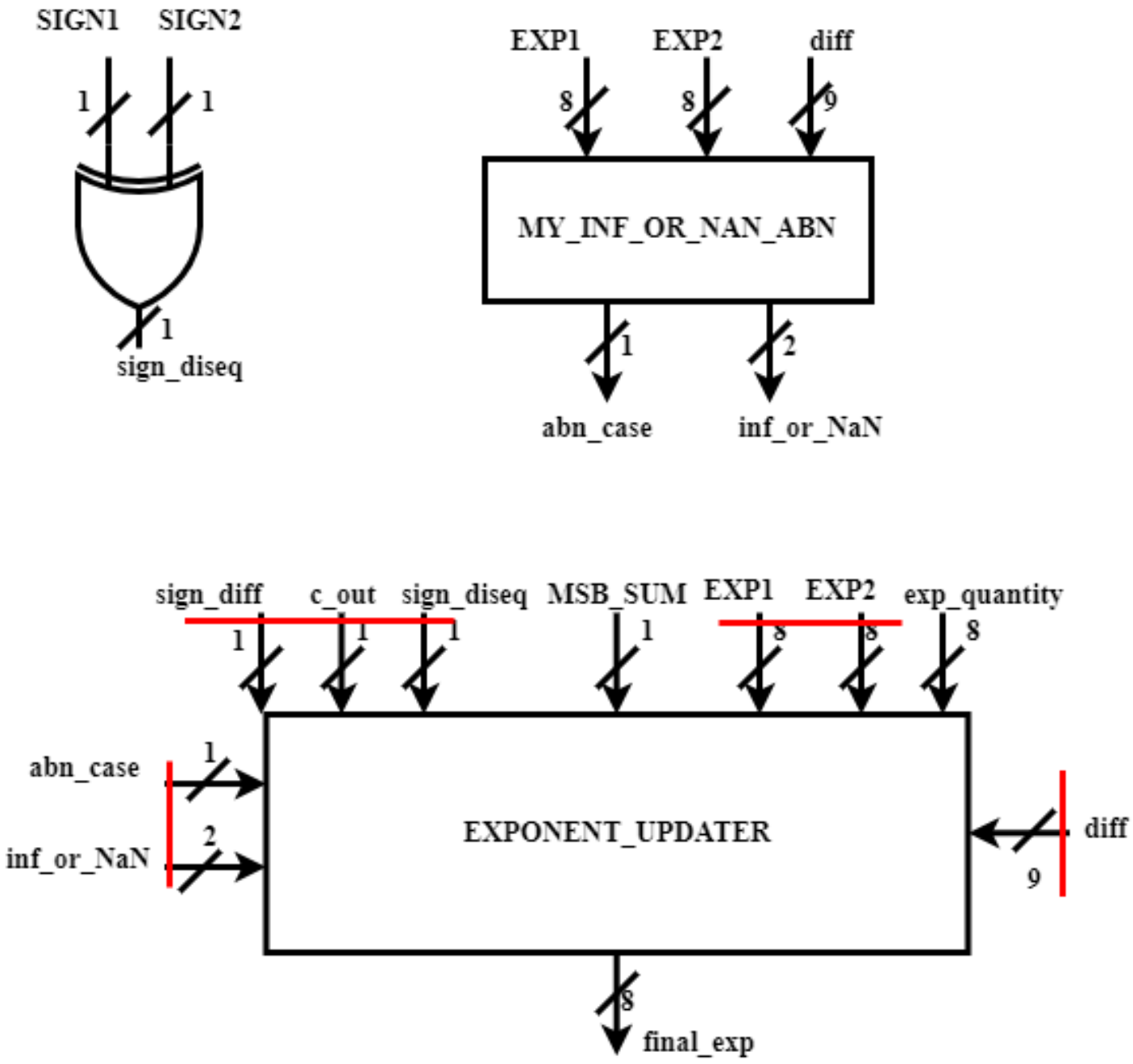

Figure 3.16: Floating-point adder with one pipeline stage(part 2)

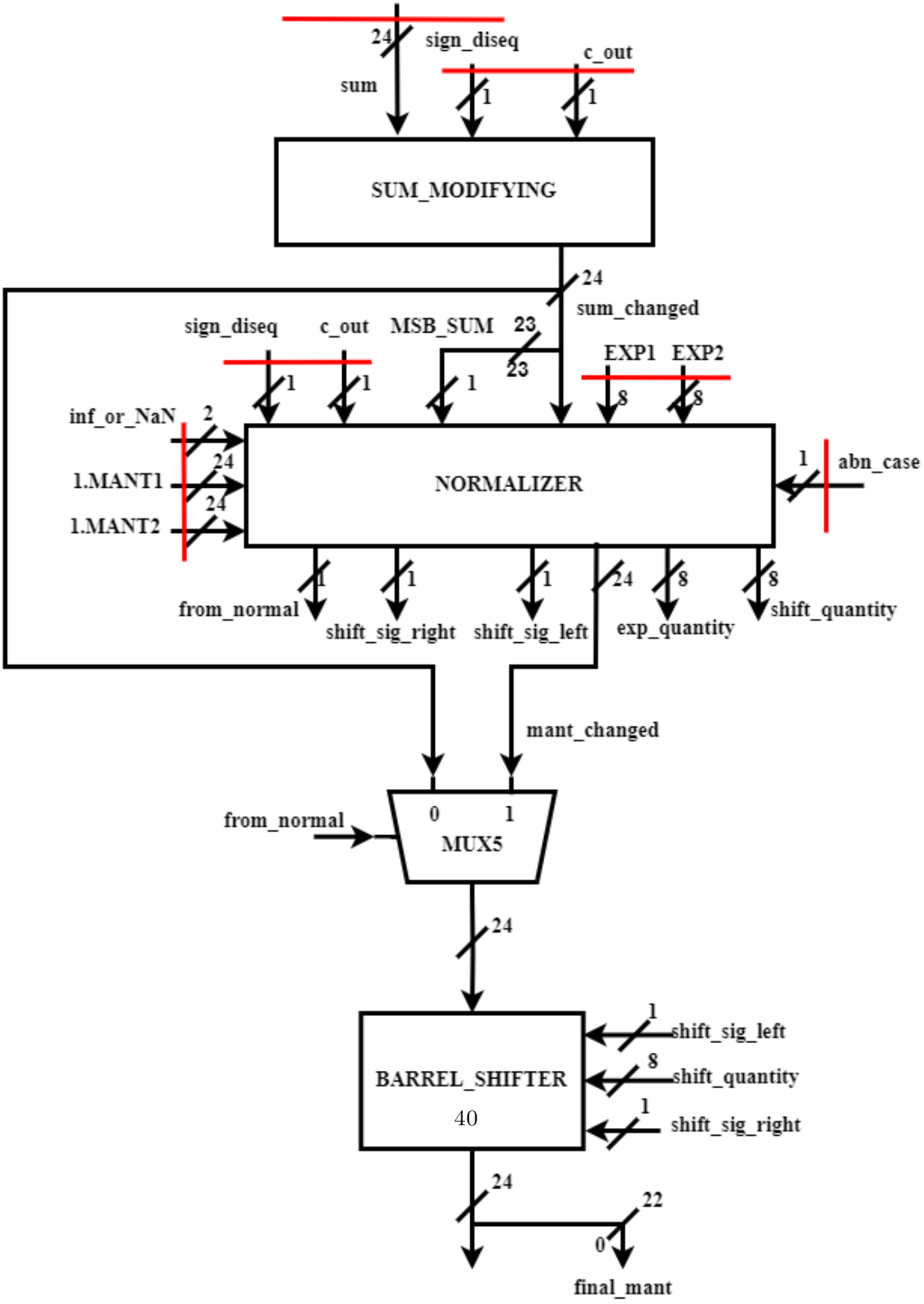

Figure 3.17: Floating-point adder with one pipeline stage(part 4)

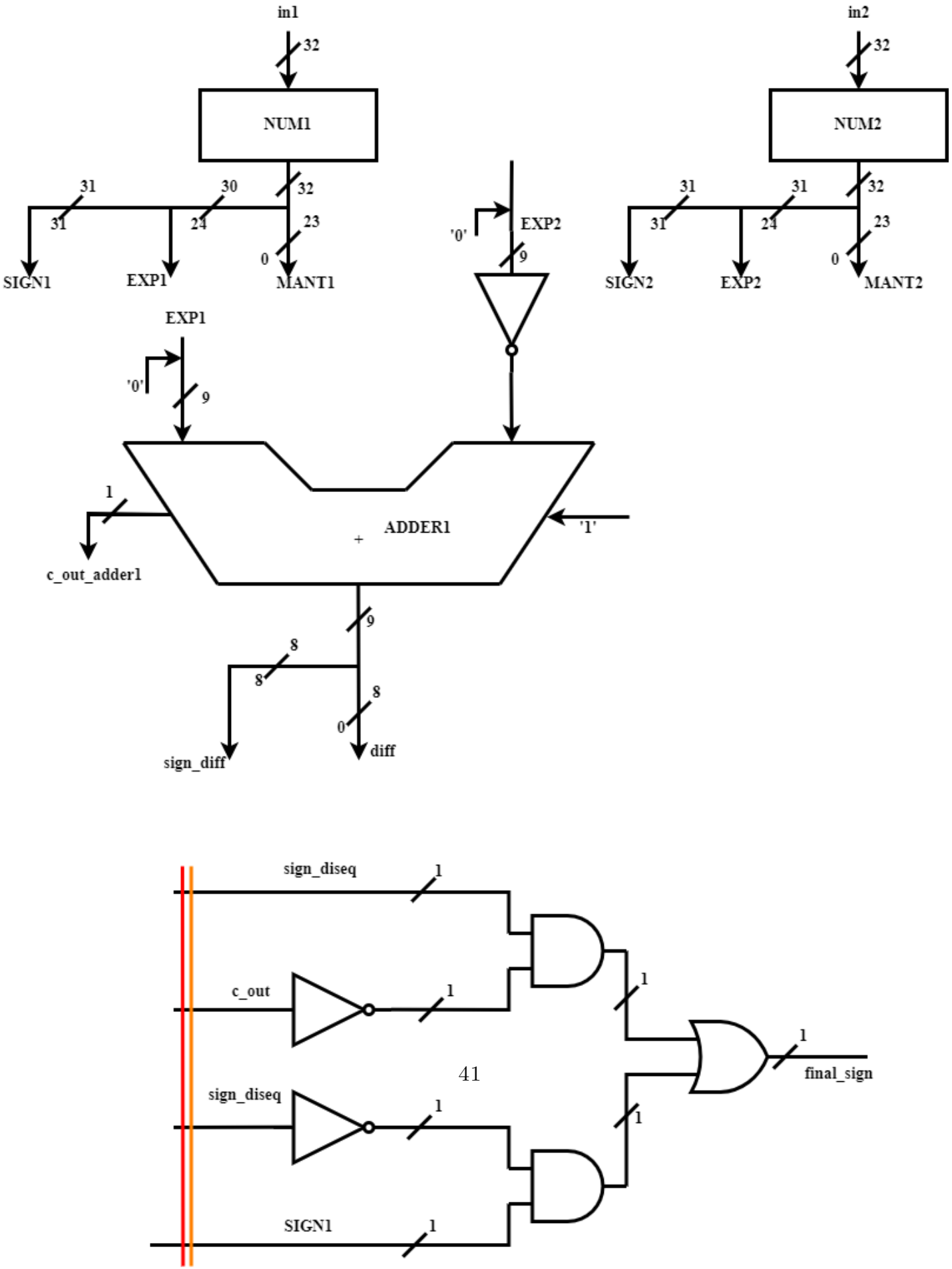

Figure 3.18: Floating-point adder with two pipeline stages(part 1)

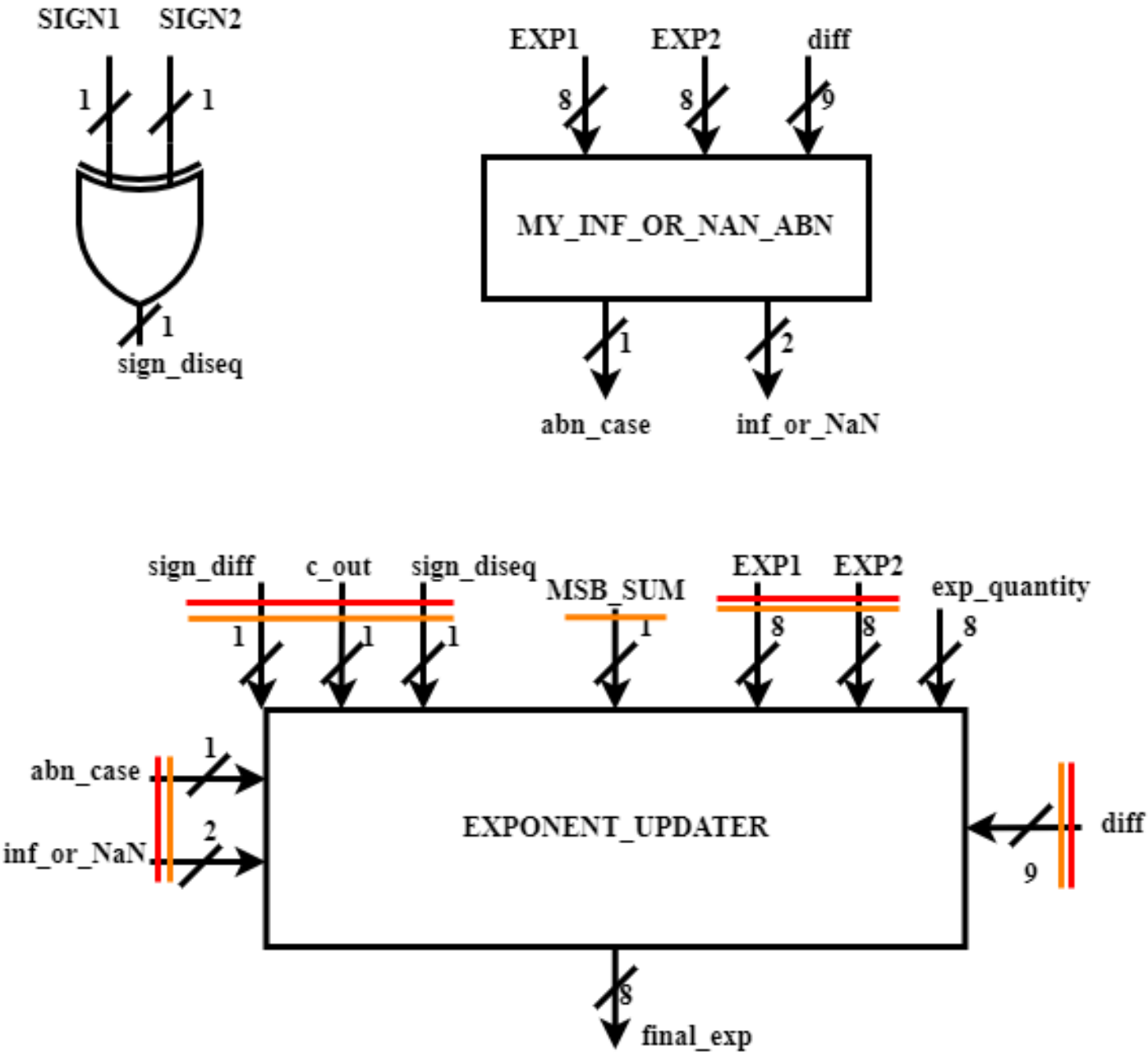

Figure 3.19: Floating-point adder with two pipeline stages(part 2)

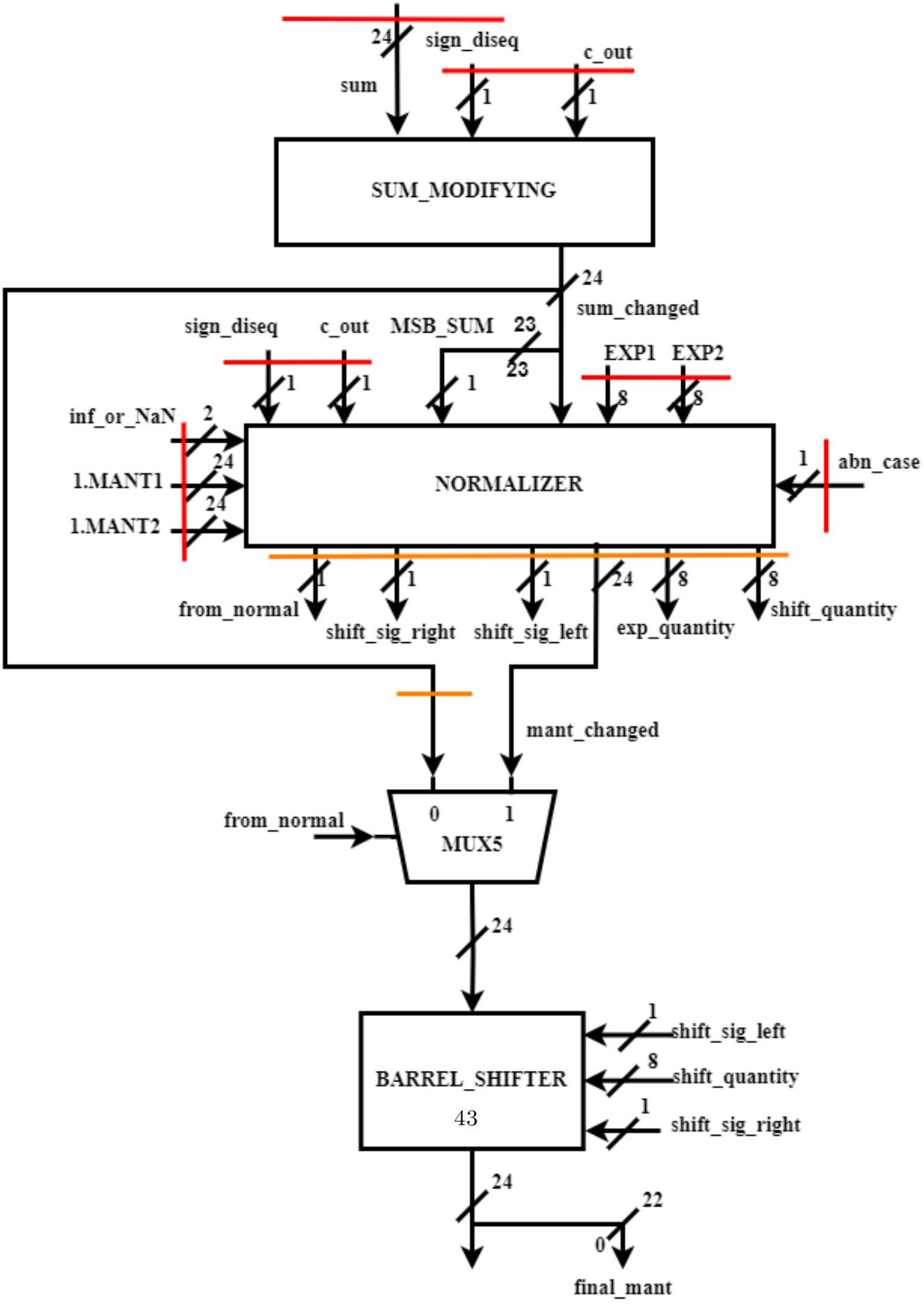

Figure 3.20: Floating-point adder with two pipeline stages(part 4)

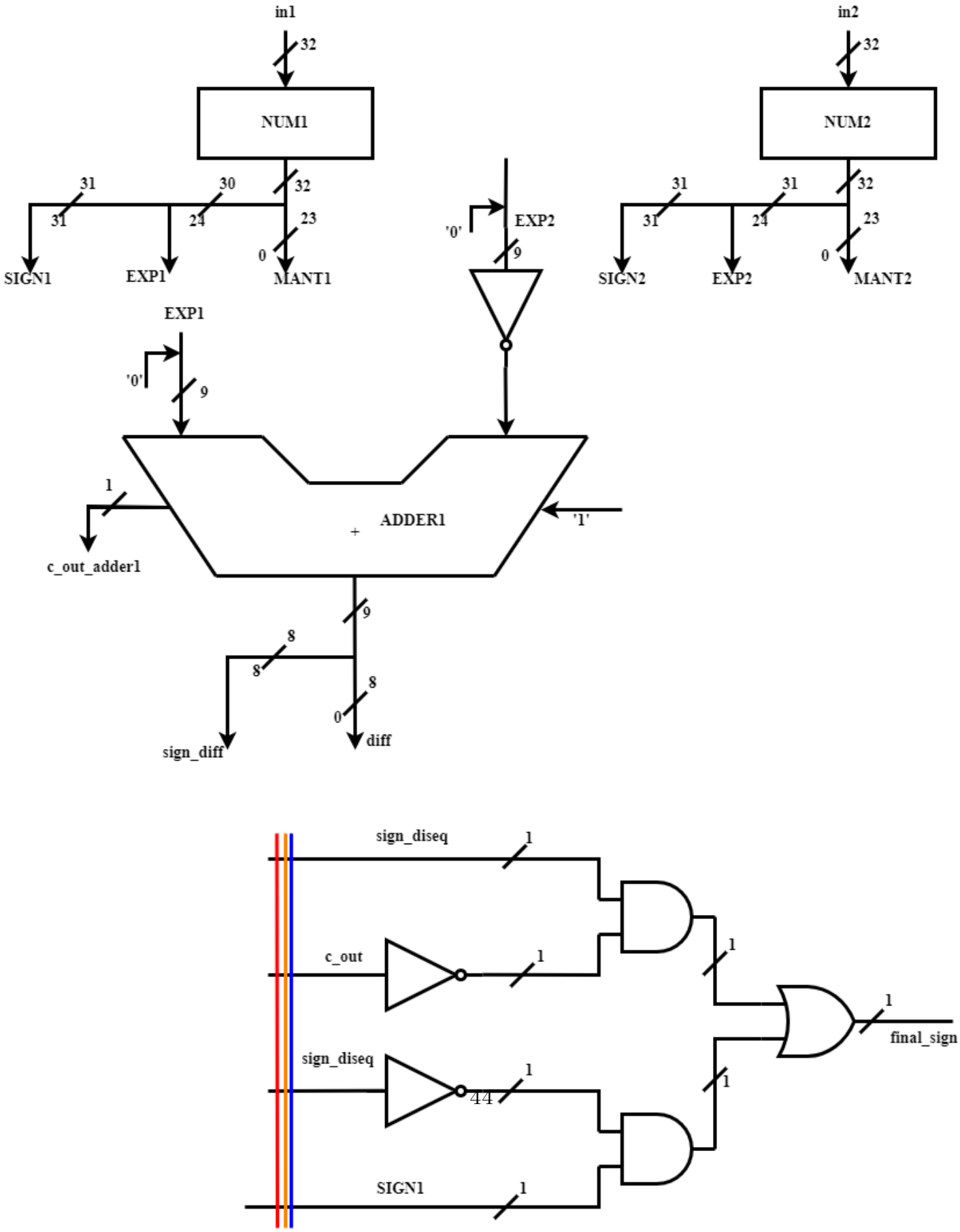

Figure 3.21: Floating-point adder with three pipeline stages(part 1)

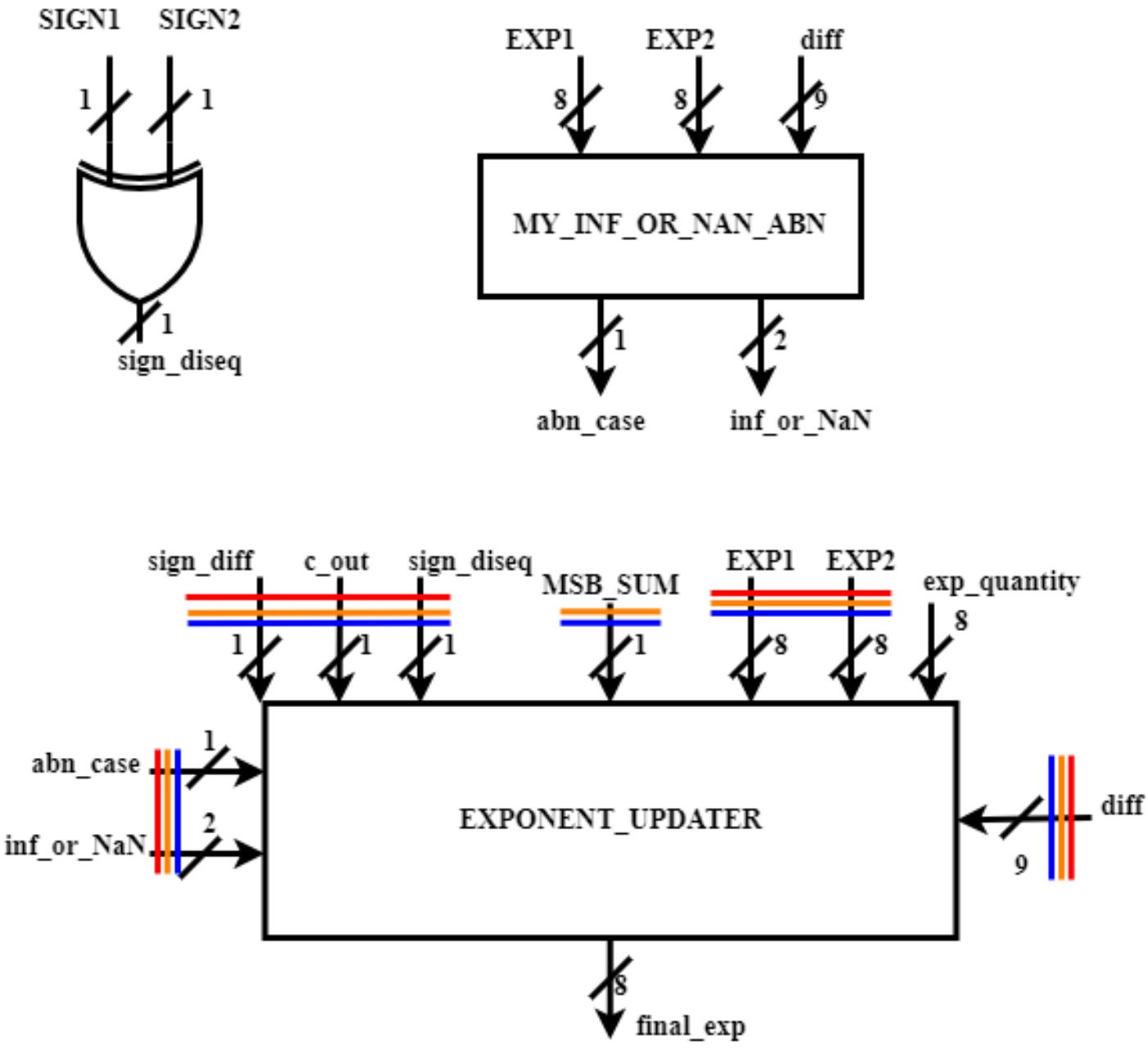

Figure 3.22: Floating-point adder with three pipeline stages(part 2)

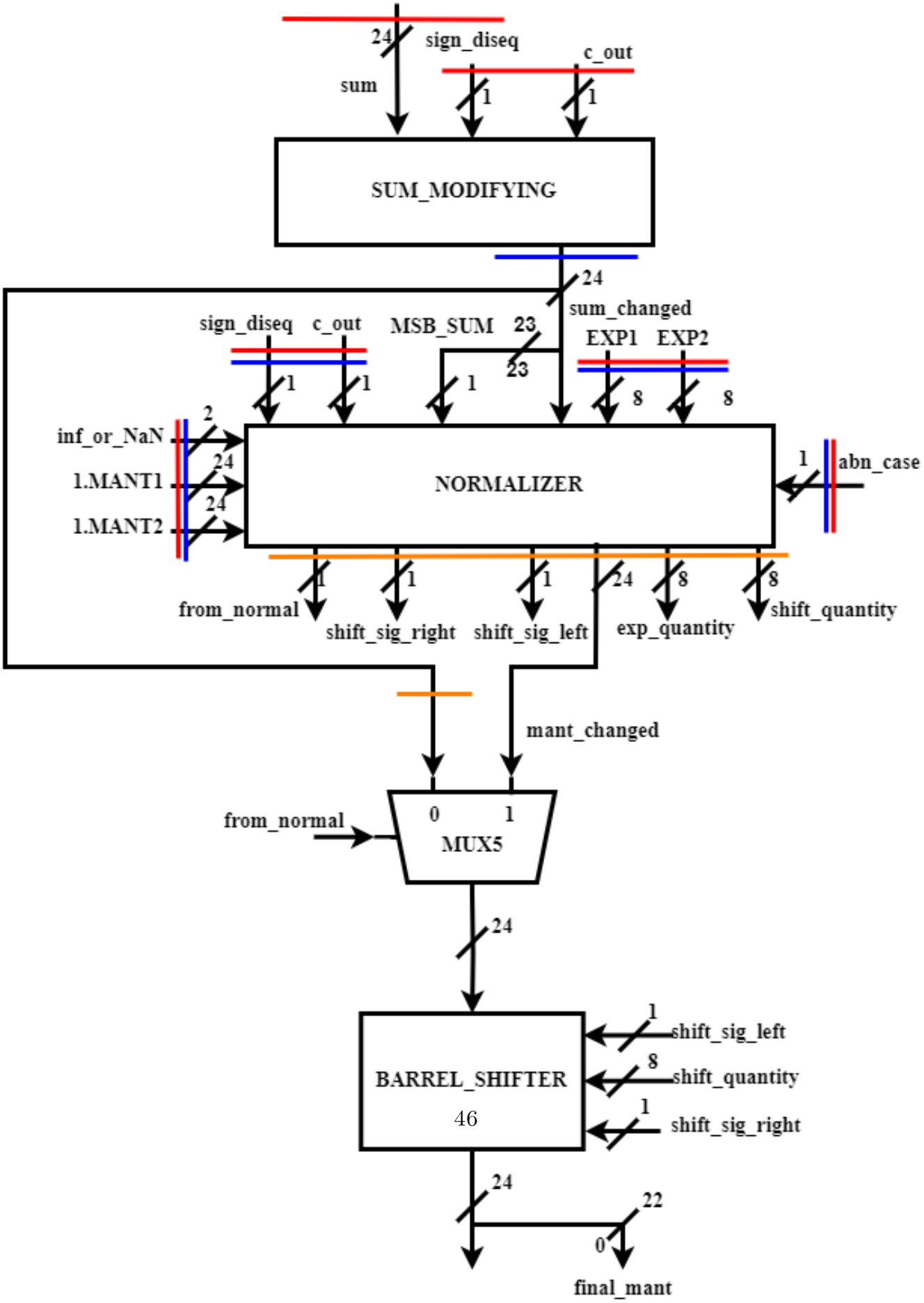

Figure 3.23: Floating-point adder with three pipeline stages(part 4)

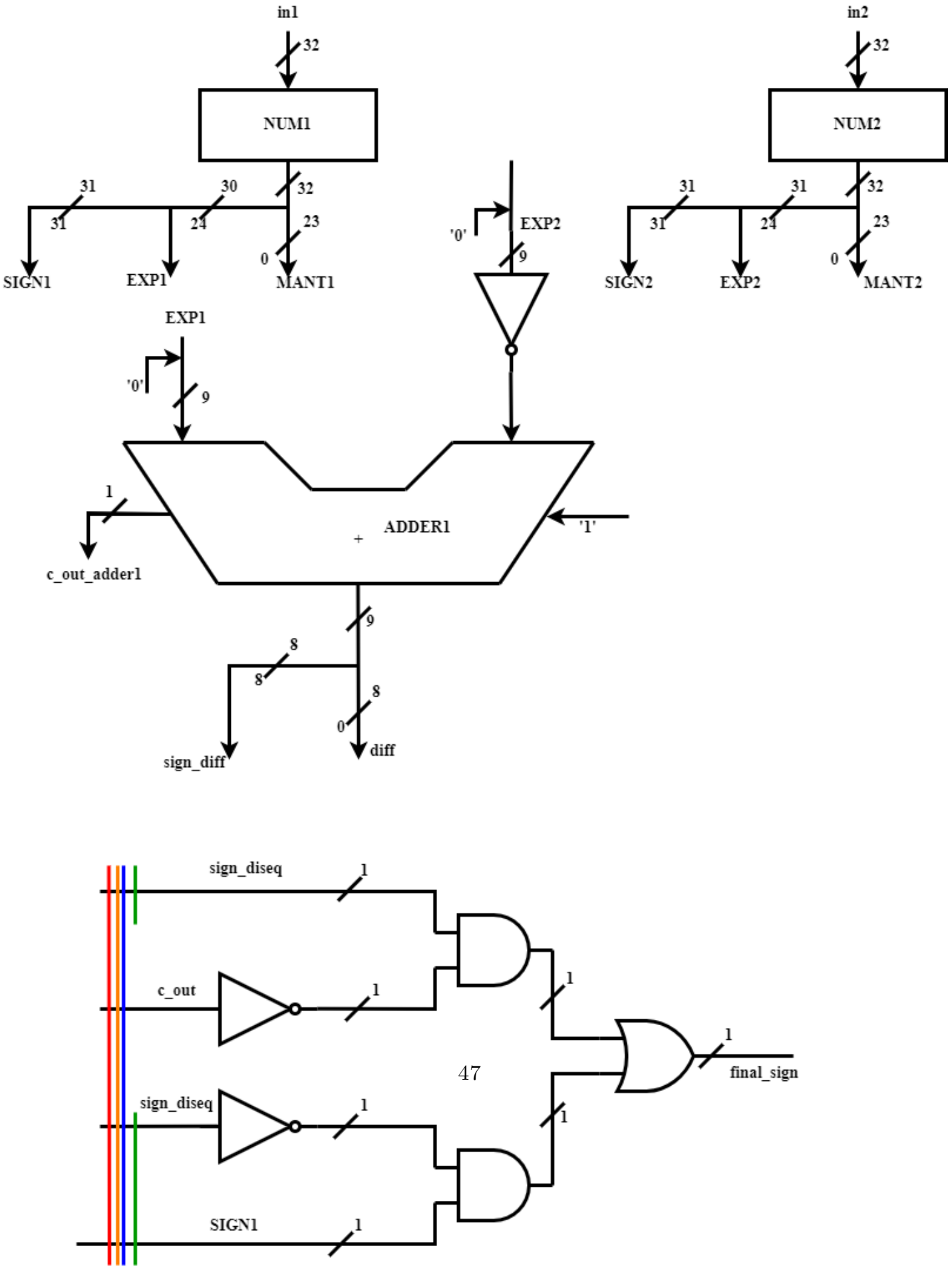

Figure 3.24: Floating-point adder with four pipeline stages(part 1)

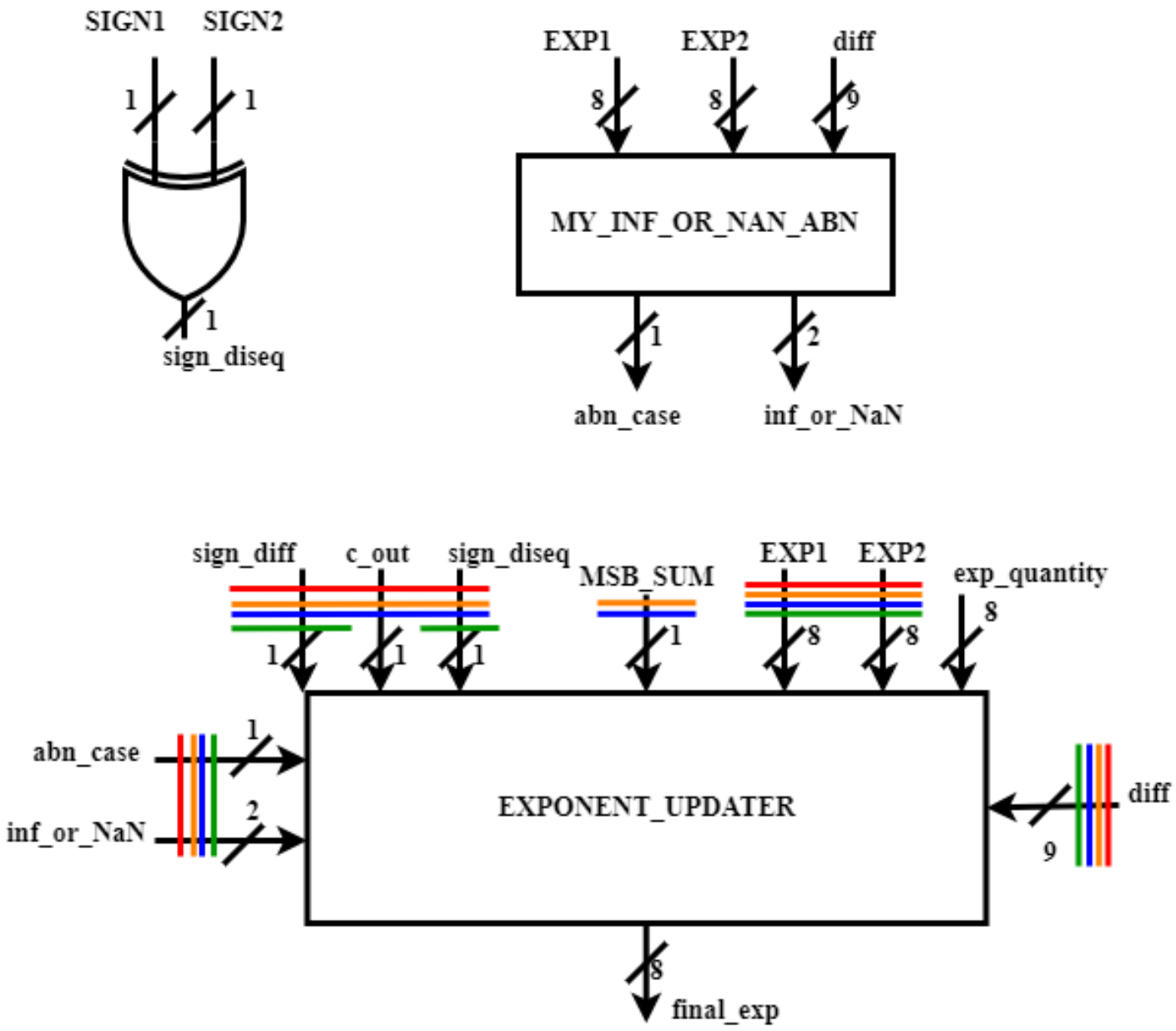

Figure 3.25: Floating-point adder with four pipeline stages(part 2)

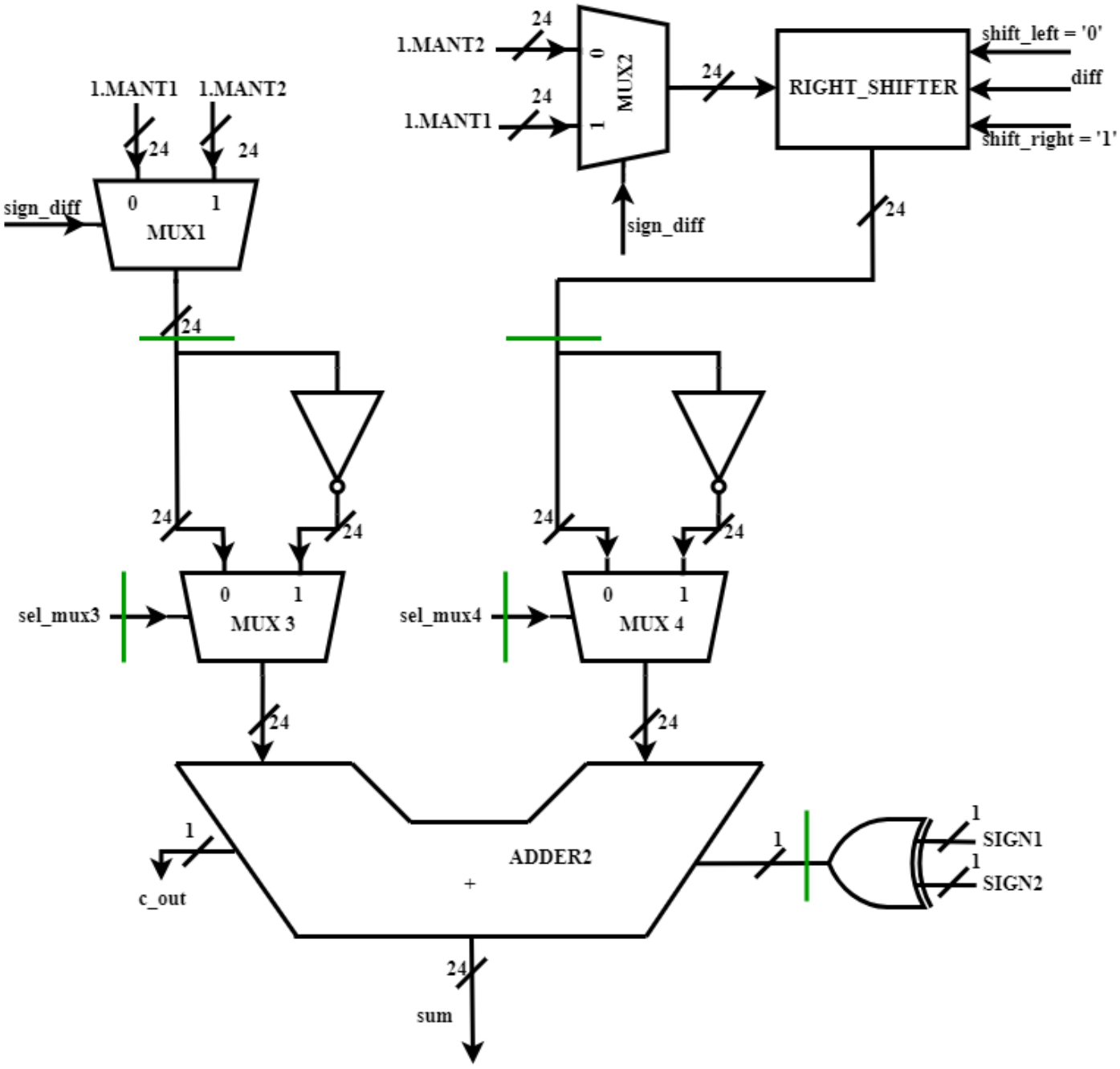

Figure 3.26: Floating-point adder with four pipeline stages(part 3)

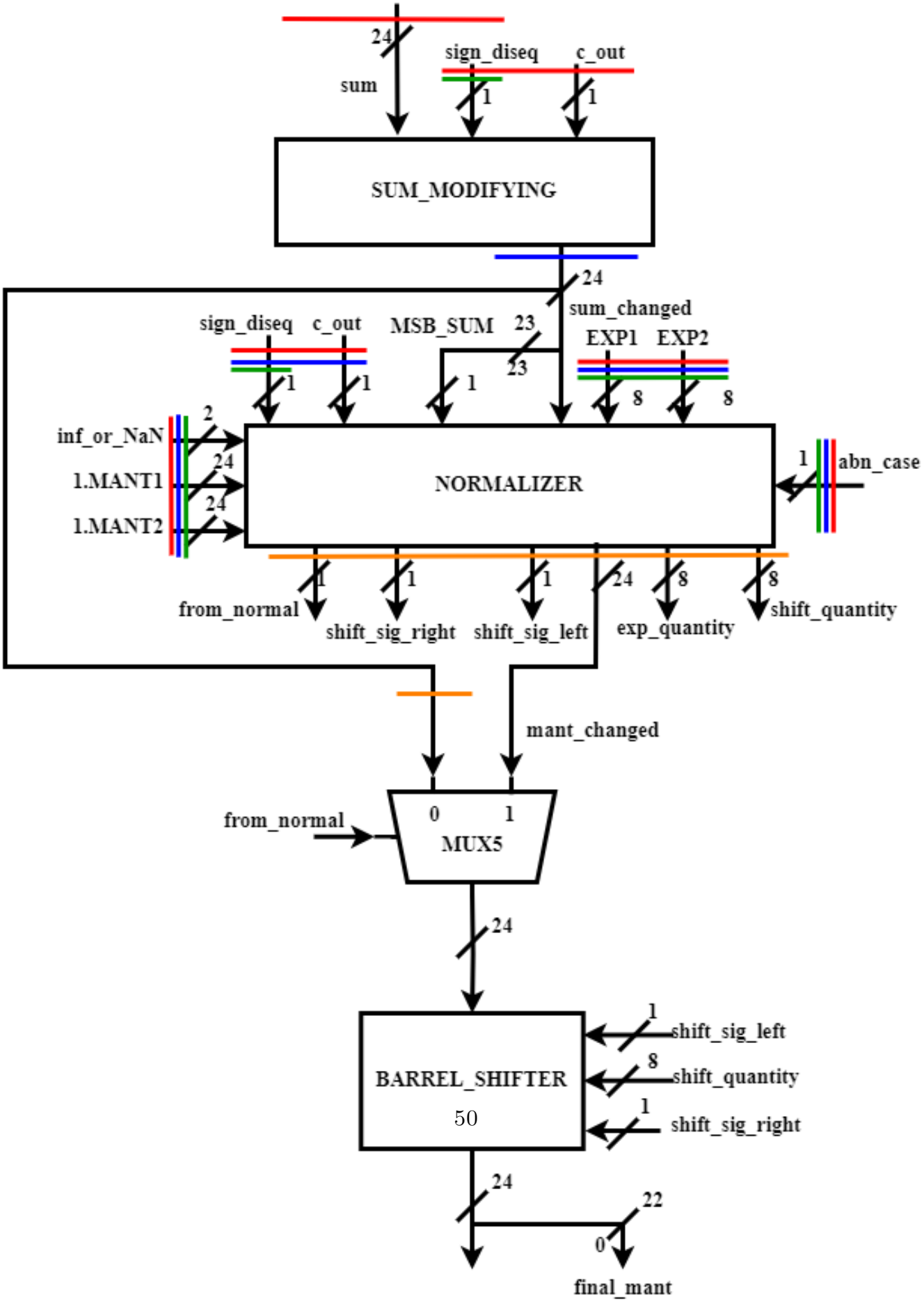

Figure 3.27: Floating-point adder with four pipeline stages(part 4)

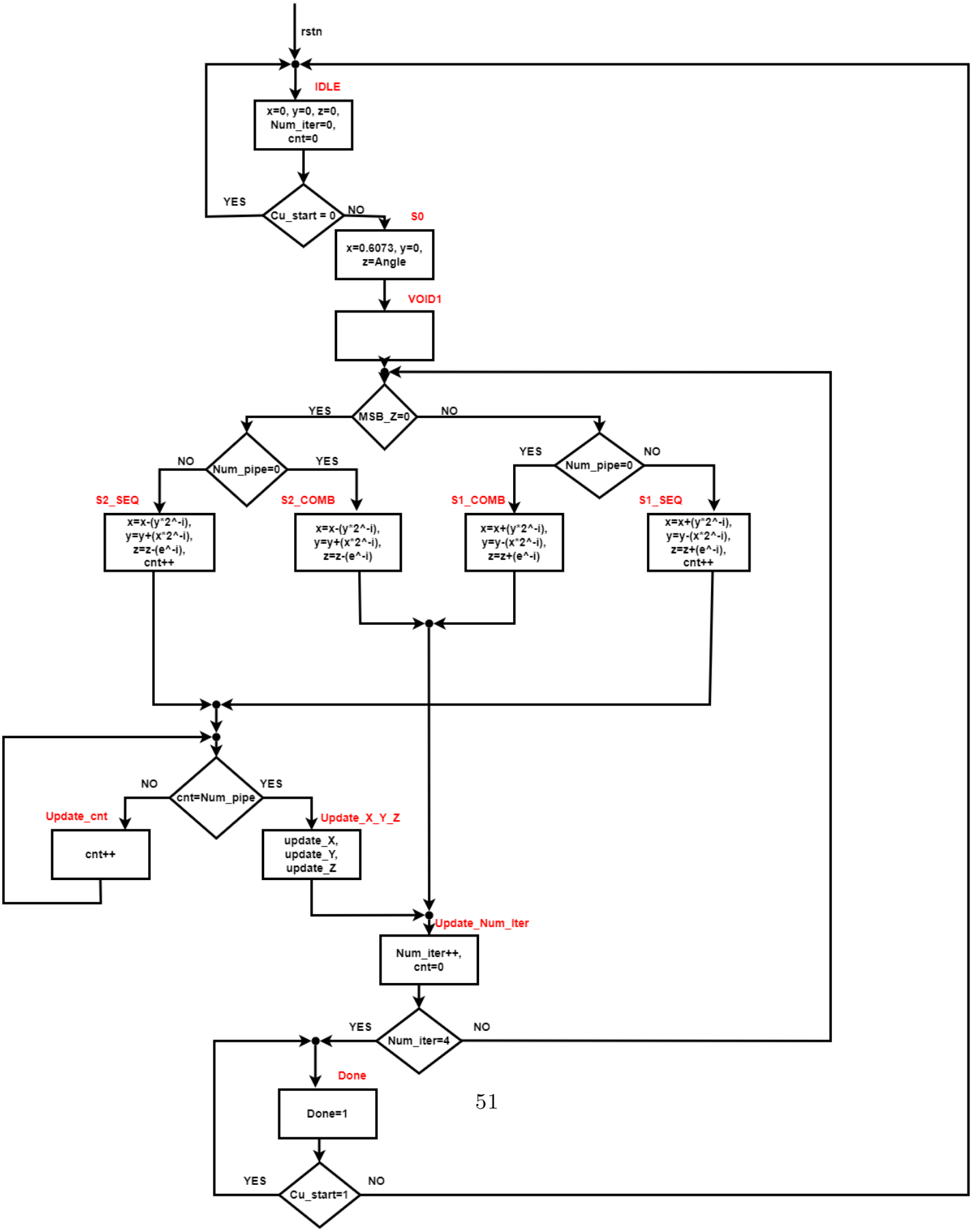

Figure 3.28: ASM chart CORDIC architecture that uses both asynchronous and pipelined floating-point adders

The CORDIC architecture that uses floating-point adders with 4 pipeline stages, being synthesized with appropriate optimizations, reaches a frequency of 970.87 MHz. Therefore the objective related to the frequency has been achieved. But this architecture has a defect, it consists of a high latency: for this type of architecture, pipelined floating-point adders are not used to their full efficiency, in other words, it is not possible to wait for N clock cycles as many as the number of pipeline stages introduced, to compute the sums in the first iteration and then begin the computation of the sums of subsequent iterations each clock cycle. This is because a given data dependency is present that cannot be resolved since each iteration starts the computations with the values of the previous iteration. So in this case the pipelined floating-point adders are useful only for reducing the critical path of the circuit, and not for doing parallel operations. In this way, considering that the CORDIC architecture with floating-point adders with 4 pipeline stages takes 37 clock cycles to compute sine and cosine functions, it would take 154 clock cycles to compute consecutively the same functions of four different values, considering that two clock cycles are required to bring the control unit start signal to logical zero and then to logical one before starting the computation of a new function. So, as previously anticipated, in order to greatly reduce the latency related to the computation of consecutive sine and cosine functions, while trying to maintain a high-frequency level, the architecture in figure 2.7 was taken as a reference. Thus, an unrolled version of the CORDIC architecture was produced.

3.5 Unrolled CORDIC architecture

In figure 3.29 is represented the Unrolled CORDIC architecture. This architecture behaves exactly like the one analyzed so far in figure 3.13. It is evident that this architecture was obtained by instantiating the CORDIC one, making some changes, a number of times equal to the number of iterations chosen, which is five. Compared with the CORDIC architecture, each instance introduces the following changes at the architecture level:

1. It is no longer necessary to use either the counter or the LUT because each instance has already fixed the number of its iteration and the corresponding value of the LUT.

2. Also the multiplexers at the input of the registers are no longer needed, since the outputs of each instance go to the input of the registers that follow them, except of course those of the last instance since two of them are the outputs of the circuit.

3. This architecture has no control unit. In fact, the load signals of registers X, Y, and Z are linked to the start signal, so that with the start equal to logical

zero, the architecture does not accept values that occur at its inputs. Instead, the remaining registers keep their load signal fixed at a logical one. The reset signal of each register corresponds to the reset signal of the entire architecture. The multiplexer selection signals of each instance are handled by the MSB of the output of its Z register.

4. Since there is no control unit, the done signal is obtained by sampling the start signal from a consecutive series of flip-flops whose number is such that the corresponding done signal arrives at the clock cycle following the one in which the architecture outputs are ready.

As with the CORDIC architecture, five versions were produced for this unrolled CORDIC architecture, each with a different type of floating-point adder. The main advantage of this architecture is the absence of the previously identified data dependency, and thus it is possible to start at each clock cycle the computation of new values for the sine and cosine functions. Considering consecutive calculations of these functions, each version of this architecture introduces both a latency reduction of 7 clock cycles compared to that obtained with the corresponding version of the CORDIC architecture for the first calculation and a latency of one clock cycle for all remaining calculations. So, taking the previous example of the CORDIC architecture that takes 154 clock cycles to calculate 4 different values of the sine and cosine functions, this architecture with the same type of floating-point adders takes 33 clock cycles. However, the main disadvantage of this architecture is related to its large area.

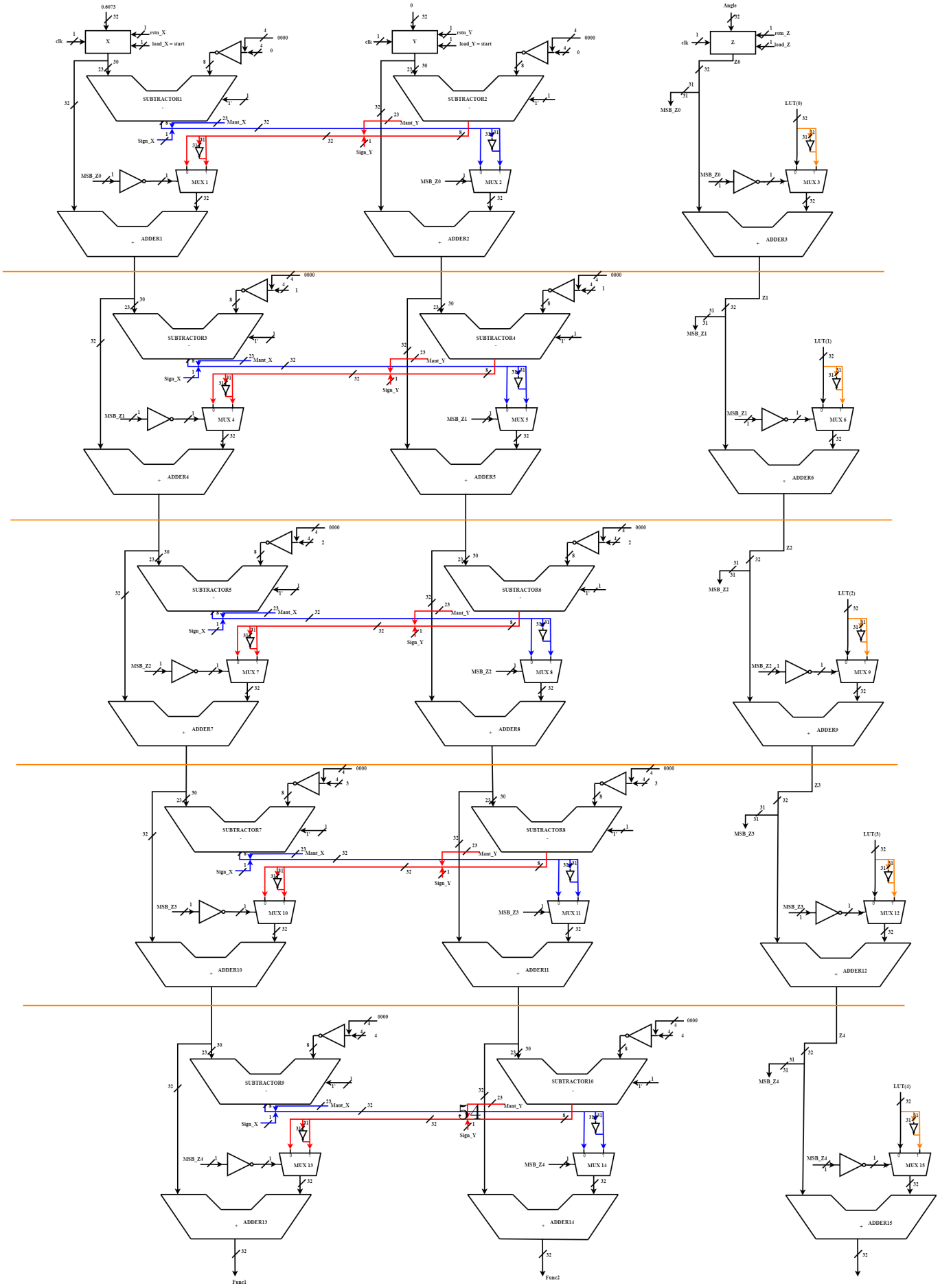

Figure 3.29: Unrolled CORDIC architecture

Chapter 4

RISC-V PULPino

The proposed designs have been tested using a RISC-V processor core. In particular PULPino, an open-source single-core microcontroller system, based on 32-bit RISCV-V cores, has been employed. Its structure is represented with a block diagram shown in figure 4.1

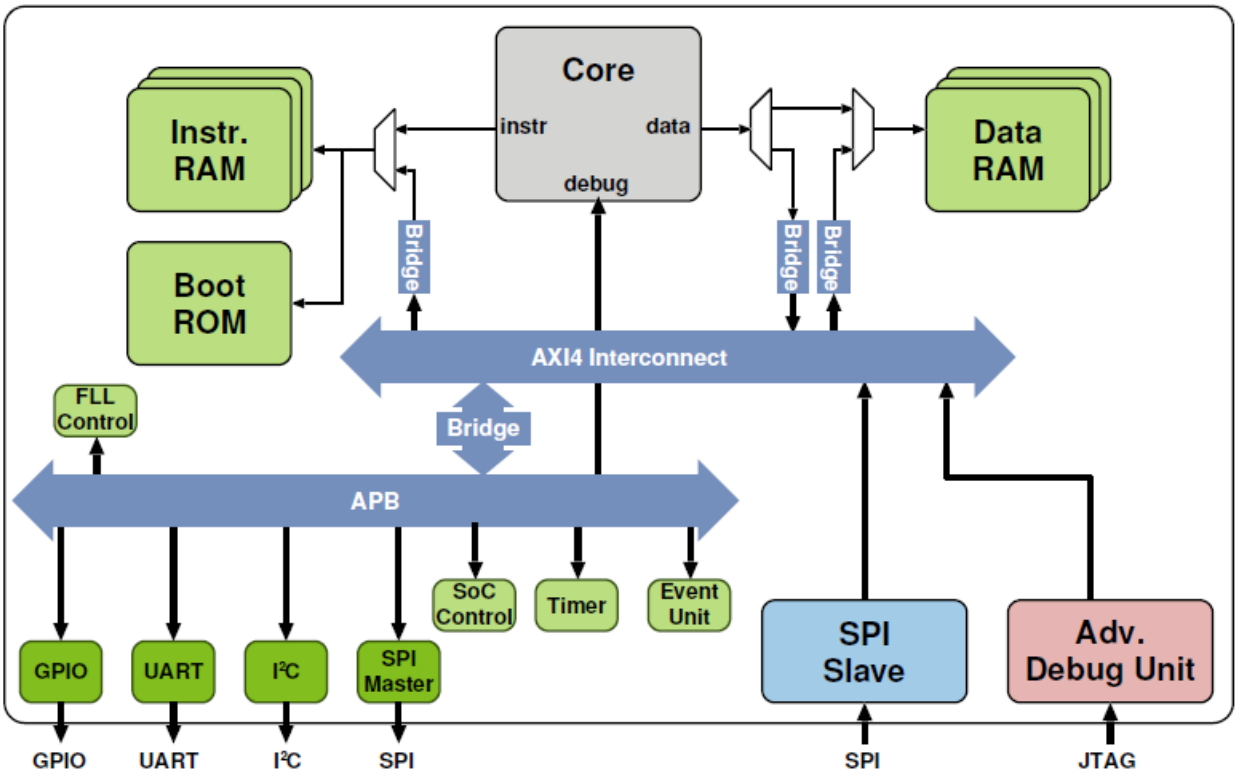

Figure 4.1: PULPino structure [12]

In this system can be used either RISCY core or zero-riscy core: RISCY core has 4 pipeline stages, and it can support integer instruction set, multiplication instruction set extension, and also compressed instructions. Moreover, it can be configured to have single-precision floating-point instruction set extension [10]; zero-riscy core has 2 pipeline stages, and it can support integer instruction set, multiplication instruction set extension, and compressed instructions. Since the RI5CY core supports floating-point extensions, it is the core chosen in this thesis work. PULPino has two separate single-port RAMs for data and instructions, and a boot ROM in which a boot loader is stored, and it can load a program from an external flash device using SPI protocol. As interconnections, there are AXI and APB buses, which have 32-bit wide data channels. The AXI type is connected by bridges to both the core and the two RAMs. Through another bridge, information on the AXI is transmitted to the APB type and vice versa. The APB type is used to connect peripherals to the system and to enable communication between the two. PULPino has peripherals such as GPIO, UART, I^2C, and SPI. Moreover, it has also an advanced debug unit, with which the core and IO memory areas can be accessed, and the two RAMs via JTAG.

4.1 Building PULPino

As described in [10], PULPino has the following requirements:

1. Modelsim in a recent version, reasonably versions equal or greater than 10.2c.
2. Python2 version equal or greater than 2.6.
3. In case Verilator is used, it is necessary that its version is 3.884.
4. CMake version equal or greater than 2.8.0.
5. Riscv-toolchain, specifically riscv32-unknown-elf-gcc compiler is needed. It is suggested to use the custom RISC-V toolchain from ETH.

Both during the building phase of PULPino and during the simulation phase of the CORDIC architecture, it was very useful to take the thesis work [13] as a reference. The toolchain was downloaded from the link [14]: the files and folders present in the link were copied into a folder using the git clone command followed by the link, and then in the same folder the command make was executed, as shown below

```
1 git clone https://github.com/pulp-platform/ri5cy_gnu_toolchain
2 make
```

After, in a new folder PULPino was downloaded from GitHub at link [10], using the git clone command again and followed by the link.

```
git clone https://github.com/pulp-platform/pulpino
```

Then, to clone git sub repositories and update them, the python file update-ips.py of PULPino was executed

```
./update-ips.py
```

In case running this file an error is generated due to the missing yaml module, it is possible to fix it by first downloading it through the following commands

```
python3 -m venv venv3.12
./venv3.12/bin/pip install pyyaml==3.12
```

and then entering its path in the python file in which it is called. In addition, running this python file, other errors may also be generated caused by the difference between the version of python in which PULPino was written and the one present on the platform on which it was downloaded. In this case, it is necessary to replace the code, indicated as an error, with the equivalent code suitable for the python version present on the platform. Then, inside the *sw* folder, the *build* folder was created. It is necessary to copy inside the *build* folder one of the four cmake-configure bash scripts that are in the *sw* folder, modify it, and run it so that it is possible to do simulations using Modelsim. These four bash scripts are the following:

1. *cmake_configure.riscv.gcc.sh* , which selects the RISCY cores and equips the system with PULP-extensions and RV32IM support. Moreover, it is necessary to check that the GCC march flag is set to "IMXpulpv2".

2. *cmake_configure.riscvfloat.gcc.sh* , which selects the RISCY cores and equips the system with PULP-extensions and RV32IMF support. Moreover, it is necessary to check that the GCC march flag is set to "IMFDXpulpv2".

3. *cmake_configure.zeroriscy.gcc.sh* , which selects the zero-riscy cores and equips the system with RV32IM support. Moreover, it is necessary to check that the GCC march flag is set to "RV32IM".

4. *cmake_configure.microriscy.gcc.sh* , which selects the zero-riscy cores and equips the system with RV32E support. Moreover, it is necessary to check that the GCC march flag is set to "RV32I".

Moreover, inside all four bash files there is RVC flag, and only by setting it equal to one it is possible to have the compressed instructions.
cmake_configure.riscvfloat.gcc.sh is the bash file chosen, because then riscy can have the floating point extensions and therefore PULPino can easily communicate with architectures, such as CORDIC, that use the floating-point format. Within this file, must be indicated the riscv32-unknown-elf-gcc compiler path, using the *export* command. Finally, inside the *build* folder, it is enough to run the following commands to complete the compilation of PULPino and the libraries it uses.

```
1 ./cmake_configure.riscvfloat.gcc.sh
2 make vcompile
```

Obviously, if vhd files are added to PULPino, as in the case of CORDIC architecture, their names have to be inserted with the command *vcom* inside *vcompile_pulpino.sh* at the path *vsim/vcompile/rtl*, before typing *make vcompile* command, since with this last command *vcompile_pulpino.sh* file is executed and consequently the vhd files are compiled. It may happen that, with the *make vcompile* command, errors may be generated, indicating that Modelsim commands such as *vmap* or *vlog* are not found. To resolve them, it is enough to indicate on the shell, using the *export* command, the path of these commands.

4.2 Connecting CORDIC architectures to PULPino with APB bus

After compiling PULPino, it was first necessary to connect CORDIC architectures to it and then begin the testing part. Since the added architectures can be considered as peripherals, a dedicated APB bus was created for them. This type of bus is composed by the following signals:

- HCLK, it is the peripheral clock.
- HRESETn, it is the asynchronous reset.
- PADDR, it is the address that will be read or written. It is on 12 bits.
- PWDATA, it corresponds to the data to be written and it is on 32 bits.
- PWRITE, with its value equal to the logical zero, it corresponds to the read signal, otherwise with its value equal to logical one it is equivalent to the write signal. This signal is synchronous.
- PSEL, it selects the peripheral, in case it is equal to the logical one.

- PENABLE, it enables the peripheral.
- PRDATA, it corresponds to the data to be read and it is on 32 bits.
- PREADY, it is the ready signal, and it indicates that the peripheral is ready to communicate or not.
- PSLVERR, it corresponds to the error signal that the peripheral sends to the core, for example in the case in which the core wants to access an address inside the peripheral where there is no register.

In order to implement it on PULPino, and thus define read and write operations aimed at the CORDIC architecture, the followed steps were followed:

1. Inside a subfolder of *rtl* folder, called *includes*, there is the file *apb_bus.sv* . In this file, the memory space dedicated to the CORDIC peripheral has been defined, and it is between the addresses 0x1A108000 and 0x1A108FFF. Moreover, the number of peripherals has been updated from 9 to 10.
2. In the file *periph_bus_wrap.sv* placed in *rtl* folder, the master associated with the new peripheral has been defined. To this master were given, as start and end addresses, the two described in the previous step.
3. Inside *apb* subfolder of *ips* folder, a new folder has been created with a dedicated file for the new APB bus in it, defining it as follows: first 7 addresses have been defined for the inputs and outputs of the CORDIC architecture, which are 0x410, 0x474, 0x4D8, 0x53C, 0x5A0, 0x604, 0x668. Each of them will be associated respectively with the input angle, the initialization value of the variable x, the initialization value of the variable y, the start signal for the architecture, the output value computed for the cosine, the output value computed for the sine, the done output signal; then for this bus were added as outputs the signals that will be sent to the CORDIC architecture, which are the angle, the initialization values of the variables x and y, and the start signal. As inputs the signals that will be received by it, which are the two functions computed by the architecture and the done signal; Successively, write operation was defined. In particular, with the peripheral ready for communication, if the signals PSEL, PENABLE, PWRITE are at the logical one, the value of PADDR is checked. If its value is equal to 0x410, the output signal corresponding to the angle will be associated with the PWDATA signal. Instead, if its value is equal to 0x474, the output signal corresponding to the initialization value of the variable x will be associated with the PWDATA signal. Or, if its value is equal to 0x4D8, the output signal corresponding to the initialization value of the variable y will be associated with the PWDATA signal. Otherwise, if its value is equal to 0x53C, the

output signal corresponding to the start signal will be associated with the LSB of PWDATA signal; finally, the read operation was defined. With the peripheral ready for communication, if the signals PSEL and PENABLE are at the logical one and PWRITE is at the logical zero, the value of PADDR is checked. If its value is equal to 0x5A0, PRDATA will be associated with the input signal corresponding to the cosine function. If its value is equal to 0x604, PRDATA will be associated with the input signal corresponding to the sine function. If its value is equal to 0x668, PRDATA will be associated with the input signal corresponding to the done signal on 32 bits.

4. The new bus was instantiated in the file *peripherals.sv*, placed in *rtl* folder.

5. In the file *pulpino_top.sv*, placed in *rtl* folder, the CORDIC architecture was instantiated, taking care to connect the outputs of the dedicated APB bus with the inputs of the CORDIC architecture, and connecting the outputs of the latter with the inputs of the APB bus.

4.3 Testing CORDIC architectures on PULPino

Since the Modelsim version used for the simulations is 64-bit, first the presence of the flag -64 was checked inside the *CMakeSim.txt* file, located in the *apps* subfolder of the *sw* folder. Moreover, to be sure that Modelsim uses the toolchain gcc and not its own, it is necessary to add to the file *vsim.tcl*, located at path *vsim/tcl_files/config*, under the line *-voptargs=+acc -suppress 2103"*, the line *$VSIM_FLAGS -dpicpppath /usr/bin/gcc"*, as shown below

```
-voptargs=\"+acc -suppress 2103\" \
$VSIM\_FLAGS -dpicpppath /usr/bin/gcc"
```

Then, inside the *apps* subfolder of *sw* folder, a new folder called *cordic* was created, with inside the C file *cordic.c* used for the test and present in the Appendix section, and also the file called *CMakeLists.txt* containing the following instruction

```
add_application(cordic cordic.c)
```

in which *cordic* is the name of the program to be run, and obviously *cordic.c* is the name of the C program to be compiled. So as to make PULPino aware of the existence of this folder and these created files, it is enough to add the name of the folder in the *CMakeLists.txt* file present inside the *apps* subfolder of *sw* folder, using *add_subdirectory* command, as shown below

```
1 add_subdirectory(cordic)
```

In order to run simulations related to the CORDIC architecture on the Modelsim GUI, it is sufficient to run the command

Chapter 5

Results and Discussion

In this section, the results of the previous work are presented. The logic synthesis tool Synopsys Design Vision and the simulation tool Modelsim were used. First, the maximum clock frequency of each floating-point adder was calculated, so as to figure out the number of pipeline stages needed to achieve a frequency on the order of 1GHz. The table 5.1 shows the periods and frequencies obtained with Synopsys and the latencies of each floating-point adder.

Floating point adders	**Period [ns]**	**Frequency [MHz]**	**Latency [cycles]**
Without pipeline stages	2.84	352.1	0
With one pipeline stage	2.08	480.8	1
With two pipeline stages	1.58	632.9	2
With three pipeline stages	1.52	657.9	3
With four pipeline stages	1.58	632.9	4

Table 5.1: Period, frequency and latency of floating point adders

From table 5.1 the floating-point adder with three pipeline stages has its maximum clock frequency larger than the frequencies of the other floating-point adders, but its value is still not close to 1GHz. Thus, the same frequencies were calculated using compile_ultra command of Synopsys, which optimizes the distribution of instantiated registers in architectures, and the corresponding results obtained are shown in table 5.2

Floating point adders	Period [ns]	Frequency [MHz]	Latency [cycles]
Without pipeline stages	2.34	427.35	0
With one pipeline stage	1.59	628.93	1
With two pipeline stages	1.33	751.88	2
With three pipeline stages	1.34	746.27	3
With four pipeline stages	1.23	813.01	4

Table 5.2: Period, frequency and latency of floating point adders with compile_ultra

From table 5.2 it is clear that the floating-point adder with four pipeline stages can reach the highest clock frequency, and its value of 813 MHz is close to the target of 1GHz set. Then also for the CORDIC architectures, with each characterized by a different type of floating-point adders, the period and the maximum clock frequency obtained with Synopsys and the latencies were calculated, shown in table 5.3

Floating point adders	Period [ns]	Frequency [MHz]	Latency [cycles]
Without pipeline stages	3.98	251.26	12
With one pipeline stage	1.81	552.49	22
With two pipeline stages	1.84	543.48	27
With three pipeline stages	1.55	645.16	32
With four pipeline stages	1.56	641.03	37

Table 5.3: Period, frequency and latency of CORDIC measured

From the table 5.3 the CORDIC architecture that uses floating-point adders with three pipeline stages reaches the highest frequency, but the target is 1GHz. For this reason the same frequencies were calculated using compile_ultra command of Synopsys, thus obtaining the results shown in table 5.4

Floating point adders	Period [ns]	Frequency [MHz]	Latency [cycles]
Without pipeline stages	2.58	387.60	12
With one pipeline stage	1.37	729.93	22
With two pipeline stages	1.35	740.74	27
With three pipeline stages	1.39	719.42	32
With four pipeline stages	1.03	970.87	37

Table 5.4: Period, frequency and latency of CORDIC obtained with compile_ultra

From the table 5.4 the CORDIC architecture that uses floating-point adders with four pipeline stages can reach a frequency on the order of 1GHz. Subsequently, the maximum clock frequency, corresponding period, and latency were also measured for each type of unrolled CORDIC architecture, obtaining the values shown in table 5.5

Level of pipe at each stage	Period [ns]	Frequency [MHz]	Latency [cycles]
0	3.55	281.69	5
1	1.61	621.12	15
2	1.93	518.13	20
3	1.56	641.03	25
4	1.61	621.12	30

Table 5.5: Period, frequency and latency of Unrolled CORDIC architecture

From the table 5.5, the architecture that uses floating-point adders with three pipeline stages achieves the highest clock frequency, but as happened with the CORDIC architecture, even with this architecture using the command compile_ultra, the Unrolled CORDIC architecture capable of operating at the highest frequency is the one that uses floating-point adders with four pipeline stages, reaching a frequency of 813MHz. Comparing the latencies of the CORDIC architectures with the latencies of Unrolled CORDIC architectures, it's clear that the latter are smaller than the former by an amount equal to 7 clock cycles. But this type of architecture has a cost in terms of area. The tables 5.6 and 5.7 show the area of CORDIC architectures and Unrolled CORDIC architectures respectively.

Floating-point adders used in CORDIC	**Area**	μm^2
Without pipeline stages	Combinational	7367.402019
	Non combinational	575.092019
	Total cell area	7942.494038
With one pipeline stage	Combinational	8796.886000
	Non combinational	3256.904105
	Total cell area	12053.790105
With two pipeline stages	Combinational	8782.522025
	Non combinational	4853.170156
	Total cell area	13635.692181
With three pipeline stages	Combinational	9578.926033
	Non combinational	6527.640211
	Total cell area	16106.566243
With four pipeline stages	Combinational	10032.722039
	Non combinational	8609.622278
	Total cell area	18642.344316

Table 5.6: Area of CORDIC architectures estimated with Synopsys

Floating-point adders used in CORDIC	**Area**	μm^2
Without pipeline stages	Combinational	35993.257992
	Non combinational	2588.712084
	Total cell area	38581.970076
With one pipeline stage	Combinational	48313.314182
	Non combinational	42860.315382
	Total cell area	91173.629565
With two pipeline stages	Combinational	41875.848130
	Non combinational	24054.912775
	Total cell area	65930.760905
With three pipeline stages	Combinational	45750.936147
	Non combinational	32458.385047
	Total cell area	78209.321194
With four pipeline stages	Combinational	48313.314182
	Non combinational	42860.315382
	Total cell area	91173.629565

Table 5.7: Area of Unrolled CORDIC architectures estimated with Synopsys

Comparing the two tables, the unrolled version occupies a total area 5 times larger than the cordic version.

Then the number of clock cycles and instructions employed by PULPino during the computation of trigonometric functions were calculated using its internal mathematical functions and not the two types of CORDIC architectures implemented, obtaining the results shown in table 5.8

Instructions in C	**Number of clock cycles**	**Number of instructions**
m1 = fSin(0)	57	45
m1 = fSin(0) m2 = fSin(1.05)	104	83
m1 = fSin(0) m2 = fSin(1.05) m3 = fSin(0.52)	151	121
m1 = fSin(0) m2 = fSin(1.05) m3 = fSin(0.52) m4 = fSin(0.78)	198	159
m1 = fSin(0) m2 = fSin(1.05) m3 = fSin(0.52) m4 = fSin(0.78) m5 = fSin(0.26)	245	197

$0 \rightarrow 0°$
$1.05 \rightarrow 60°$
$0.52 \rightarrow 30°$
$0.78 \rightarrow 45°$
$0.26 \rightarrow 15°$

Table 5.8: Numbers of clock cycles and instructions using fSin (Pulpino function)

From table 5.8 it is clear that by increasing the number of functions to be computed, the number of clock cycles and instructions used for the computation increases accordingly. Finally, the number of clock cycles and instructions employed by PULPino during the computation of the same trigonometric functions using both types of CORDIC architectures were calculated. In particular, PULPino employs the same number of instructions both using the CORDIC architecture and using the Unrolled CORDIC architecture, whose values are shown in table 5.9

Pipe levels per stage	Clock cycles	Instructions for each set of C instructions
0	5	24, 35, 45, 55, 65
1	15	26, 38, 48, 58, 68
2	20	26, 38, 48, 58, 68
3	25	28, 41, 51, 61, 71
4	30	28, 41, 51, 61, 71

Table 5.9: Numbers of clock cycles and instructions using CORDIC architecture in Pulpino with the equivalent C commands seen in the previous table

As represented in table 5.9, the number of instructions obtained for each set of trigonometric function calculations is smaller than the corresponding value obtained by PULPino with its internal functions. So, the target of implementing a CORDIC architecture with which PULPino can employ fewer instructions than those obtained through the use of its internal functions has been fully achieved. On the other side, regarding the number of clock cycles employed using the CORDIC architecture, it is not possible to calculate them using the performance counters of PULPino, since it takes 9 clock cycles to write two consecutive angles to the CORDIC architecture, not one. So the values given in the table 5.9refer to the latency of Unrolled CORDIC architecture, but the latter architecture can be considered very good in terms of latency, since with the exception of the first calculated, all the other functions are ready at each clock cycle. But it is necessary to pay a cost in terms of area.

Chapter 6

Conclusions

With this thesis work, first the CORDIC algorithm was presented at the mathematical level, and then its possible implementations were analyzed. Among them, two were investigated in depth: the first one was adapted to floating-point format and then modified in order to have high frequency; the second one was the unrolled version of the first implementation, and it is used to have a reduced latency. Both types of architectures can work up to a maximum frequency on the order of 1GHz. In addition, it has been shown that a system such as PULPino, using these types of architectures instead of its internal functions, employs fewer instructions during the computation of trigonometric functions such as sine and cosine. Finally, the Unrolled CORDIC architectures can be considered valid alternatives to the PULPino functions in case the target consists to have a reduced latency but being ready to pay costs in terms of area.

6.1 Future works

This thesis work can be considered a good starting point for possible future works: one of these could aim to resolve the data dependency present in the first architecture implemented. In this way an architecture capable of working at high frequency, with reduced latency and also optimized in terms of area would be implemented; another work could be to optimize the proposed architectures in terms of power. Both architectures described so far contain registers used for pipeline, that are always active, and therefore they negatively affect the power consumption. By adopting low-consumption techniques, the power consumed could be reduced. Another work could also be to implement a control architecture that can manage CORDIC architecture in the calculation of each type of function, in order to have a unique architecture that is capable of calculating any transcendental function.

Appendix A

Listing A.1: C code used to compute the number of instructions employed during the computation of five sine functions using CORDIC architectures.

```
#include "bench.h"

#define REG_ANGLE        0x1A108410
#define REG_X            0x1A108474
#define REG_Y            0x1A1084D8
#define REG_START        0x1A10853C
#define REG_FUNC1        0x1A1085A0
#define REG_FUNC2        0x1A108604
#define REG_DONE         0x1A108668

int main()
{
  float m1, m2, m3, m4, m5;
  cpu_perf_conf_events(SPR_PCER_EVENT_MASK(1));
  perf_reset();
  *(volatile float*) REG_ANGLE = 0.0;
  *(volatile float*) REG_X = 0.6073;
  *(volatile float*) REG_Y = 0.0;
  *(volatile int*) REG_START = 1;
  while(*(volatile int*) REG_DONE != 1){
  }
  m1 = *(volatile float*) REG_FUNC2;

  *(volatile float*) REG_ANGLE = 60.0;
  while(*(volatile int*) REG_DONE != 1){
  }
  m2 = *(volatile float*) REG_FUNC2;
  *(volatile float*) REG_ANGLE = 30.0;
  while(*(volatile int*) REG_DONE != 1){
  }
  m3 = *(volatile float*) REG_FUNC2;
   *(volatile float*) REG_ANGLE = 45.0;
  while(*(volatile int*) REG_DONE != 1){
```

```
34    }
35    m4 = *(volatile float*) REG_FUNC2;
36     *(volatile float*) REG_ANGLE = 15.0;
37    while(*(volatile int*) REG_DONE != 1){
38    }
39    m5 = *(volatile float*) REG_FUNC2;
40    perf_stop();
41    printf("Number of instructions : %d\n", cpu_perf_get(1));
42 return 0;
43 }
```

Listing A.2: C code used to compute the number of clock cycles employed during the computation of five sine functions using fSin function of PULPino.

```
#include "bench.h"
#include "math_fns.h"

int main()
{

  double m1, m2, m3, m4, m5;
  cpu_perf_conf_events(SPR_PCER_EVENT_MASK(0));
  perf_reset();
  m1 = fSin(0);
  m2 = fSin(1.05);
  m3 = fSin(0.52);
  m4 = fSin (0.78);
  m5 = fSin (0.26);
  perf_stop();
  printf("Clock cycles : %d\n", cpu_perf_get(0));*/
return 0;
}
```

Listing A.3: C code used to compute the number of instructions employed during the computation of five sine functions using fSin function of PULPino.

```
#include "bench.h"
#include "math_fns.h"

int main()
{

  double m1, m2, m3, m4, m5;
  cpu_perf_conf_events(SPR_PCER_EVENT_MASK(1));
  perf_reset();
  m1 = fSin(0);
  m2 = fSin(1.05);
  m3 = fSin(0.52);
  m4 = fSin (0.78);
  m5 = fSin (0.26);
  perf_stop();
  printf("Number of instructions : %d\n", cpu_perf_get(1));*/
return 0;
}
```

Bibliography

[1] Shoaib Bhuria and P Muralidhar. «FPGA implementation of sine and cosine value generators using Cordic Algorithm for Satellite Attitude Determination and calculators». In: *2010 International Conference on Power, Control and Embedded Systems*. IEEE. 2010, pp. 1–5 (cit. on pp. 1, 2, 13, 14).

[2] Debaprasad De, Archisman Ghosh, K Gaurav Kumar, Anurup Saha, and Mrinal Kanti Naskar. «Multiplier-less hardware realization of trigonometric functions for high speed applications». In: *2018 IEEE Applied Signal Processing Conference (ASPCON)*. IEEE. 2018, pp. 149–152 (cit. on pp. 1, 2, 13, 14).

[3] Recep Onur Yıldız and Ayse Yilmazer-Metin. «CORDIC Accelerator for RISC-V». In: *2021 29th Telecommunications Forum (TELFOR)*. IEEE. 2021, pp. 1–4 (cit. on pp. 1, 13, 14).

[4] Nasrallah Tahir, Samir Tagzout, and Ahmed Amine El Ouchdi. «CAMs and high speed high precision data for trigonometric functions». In: *2015 9th International Conference on Electrical and Electronics Engineering (ELECO)*. IEEE. 2015, pp. 1169–1172 (cit. on pp. 1, 13, 14).

[5] Anurup Saha, K Gaurav Kumar, Archisman Ghosh, and Mrinal Kanti Naskar. «Area efficient architecture of Hyperbolic functions for high frequency applications». In: *2017 International Conference on Circuits, Controls, and Communications (CCUBE)*. IEEE. 2017, pp. 139–142 (cit. on pp. 1, 2, 13, 14).

[6] Srikala Vadlamani and Wagdy Mahmoud. «Comparison of CORDIC algorithm implementations on FPGA families». In: *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No. 02EX540)*. IEEE. 2002, pp. 192–196 (cit. on pp. 1, 2, 13, 14).

[7] Puli Anil Kumar. «FPGA implementation of the trigonometric functions using the CORDIC algorithm». In: *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*. IEEE. 2019, pp. 894–900 (cit. on pp. 1, 2, 13–15).

[8] Chen Dong, Chen He, Sun Xing, and Pang Long. «Implementation of single-precision floating-point trigonometric functions with small area». In: *2012 International Conference on Control Engineering and Communication Technology*. IEEE. 2012, pp. 589–592 (cit. on pp. 1, 13, 14).

[9] Kuan Jen Lin and Chien Chih Hou. «Implementation of trigonometric custom functions hardware on embedded processor». In: *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*. IEEE. 2013, pp. 155–157 (cit. on pp. 1, 13, 14).

[10] https://github.com/pulp-platform/pulpino (cit. on pp. 2, 56, 57).

[11] Leonardo Pedone. «Design and characterization of Variable Latency adders for floating-point arithmetic units». Apr. 2018. URL: `http://webthesis.biblio.polito.it/7439/` (cit. on p. 8).

[12] https://github.com/pulp-platform/pulpino/blob/master/doc/datasheet/datasheet.pdf (cit. on p. 55).

[13] Alessandro Bruscia. «Turbo decoding algorithm parallelization». Apr. 2019. URL: `http://webthesis.biblio.polito.it/10983/` (cit. on p. 56).

[14] https://github.com/pulp-platform/ri5cy_gnu_toolchain (cit. on p. 56).