Hardware Implementation of Modal Interval Adder/Subtractor and Multiplier

by

Eng. Ayman Abd-ElAziz Bakr Omar
Electronics and Communications Department
Faculty of Engineering, Cairo University

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirement for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

Hardware Implementation of Modal Interval Adder/Subtractor and Multiplier

by

Eng. Ayman Abd-ElAziz Bakr Omar
Electronics and Communications Department
Faculty of Engineering, Cairo University

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirement for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Under the supervision of

Associate Prof. Hossam A. H. Fahmy
Electronics and Communications Dept.

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

Hardware Implementation of Modal Interval Adder/Subtractor and Multiplier

by

Eng. Ayman Abd-ElAziz Bakr Omar
Electronics and Communications Department
Faculty of Engineering, Cairo University

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirement for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Approved by the
Examining Committee

Associate Prof. Hossam A. H. Fahmy , Thesis Main Advisor

Prof. Dr. Amin M. Nassar, Member

Prof. Dr. Ashraf M. F. Salem, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2012

Acknowledgements

First of all, I would like to thank Allah for helping me, supporting me, giving me the strength in hard times and for all unlimited graces he is giving to me. Secondly, I would like to thank my parents who took care of me all these years and was encouraging me to complete my thesis. Lastly, I would like to thank Prof. Hossam for his kindness and patience with me. He was like a big brother to me who advised me and taught me lots of things that I had never learnt without his help.

Abstract

Rounding errors in digital computations using floating point numbers may result in totally inaccurate results. One of the mathematical solutions to monitor and control rounding errors is the classical interval arithmetic (CIA). A generalized extension of the classical intervals was presented in 1980 which is the modal intervals. Modal Intervals Arithmetic (MIA) proved to be a good tool in many branches of applied mathematics. This leads to solving serious problems in applications like control and computer graphics. The increasing demand of high speed applications and in the same time accurate results lead researches to hardware implementation of MIA.

This work introduces, for the first time, the hardware implementation of the Modal Interval Double Floating Point Adder/Subtractor and Multiplier units. It proposes two different hardware implementation approaches (serial and parallel) for each of these units. Serial implementations have smaller areas than those of the parallel implementations but they are slower than parallel implementations. Also, there is no overhead in supporting modal intervals instead of supporting classical intervals only. Moreover, certain modal interval implementations have smaller areas than their classical counterparts. Accordingly, modal interval Adder/Subtractor and multiplier units are more efficient than their classical counterparts.

The application nature and the cost are the major benchmarks that determine whether the serial approach (smaller but slower) or the parallel approach (bigger but faster) is suitable.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

1.1 Overview

One of the main shortages of the floating point representation in digital systems is the rounding [1]. The rounding can result in catastrophic errors which make the floating point computations result in a totally different result from the exact one. Consider for example, $\mathrm{b} = 3.34, \mathrm{a} = 1.22, \text{and } \mathrm{c} = 2.28$. The exact value of $\mathrm{b}^2 - 4\mathrm{ac} = 0.0292$. But (in a system with three decimal places) b^2 rounds to 11.2 and 4ac rounds to 11.1, hence the final answer is 0.1 which is an error by 700 ulps, even though 11.2 - 11.1 is exactly equal to 0.1 . The subtraction did not introduce any error, but rather exposed the error introduced in the earlier multiplications [1]. The previous example is a simple one but we can have more complex examples that yield totally wrong results as in the following arithmetic expression [37]:

$$f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

For $a = 77617 \; and \; b = 33096$, Using IEEE 754 arithmetic operations with a round-to-nearest rounding we have the following results for 32, 64 and 128 floating point representations:

32-bit: $f \approx 1.172604$

64-bit: $f \approx 1.1726039400531786$

128-bit: $f \approx 1.172603940053178631858834904520 1838$

According to [37] and [38], the true value is $f = -0.827386\ldots$

We have these wrong results in different precisions due to the heavy cancellation in the floating point computations. The more important in the

previous example that no matter we use higher floating point precision arithmetic we may still have wrong results.

Due to this inaccuracy of floating point computations, several numerical methods are used to account for rounding errors in floating point computations or even eliminate them. One of these numerical methods is the interval computations which was greatly popularized as classical interval arithmetic by Ramon E.Moore in 1966 [2]. The basic idea of the classical intervals is to use intervals instead of real numbers. For example, the number 3.145 lies in the interval [3.13, 3.15]. Thus when we represent the classical interval bounds (3.13 and 3.15) using floating point numbers and apply the different floating point computations on it, we can monitor and control the rounding errors in the floating point computations. Also this approach allows us to represent the uncertainty of the measurements of the different systems. Consider for example, that we want to measure the temperature of a certain system. Instead of representing the temperature value with a single uncertain value, we can represent it with an interval that we are sure that the real system's temperature lies in it [2]. After the definition of classical interval arithmetic (or set-theoretical interval arithmetic) many mathematicians tried to extend it to eliminate some of its problems. One of these extensions is the modal interval arithmetic [5].

Several software packages were introduced to implement many forms of interval computations. Many interval applications need not only the accurate computations but also to be as fast as the floating point computations. The increasing need to have interval computations with a performance comparable to floating point computations leads us to the hardware implementation of the interval computations [3], [4]. Hardware implementation of the interval addition, subtraction and multiplication is what we will discuss.

1.2 Classical Intervals

1.2.1 Historical Background

The history of intervals in general goes back to the very first publications on the topic of interval calculus. There are two papers considered as the pioneering works in this field: one by the mathematician T. Sunaga in 1958 [32], and another by M. Warmus in 1956 [33]. Both were apparently completed independent of each other. In 1961, a second paper appeared by Warmus [34]. In the paper by Sunaga, almost all foundational elements of the interval calculus, as known today, are presented [6].

Since the publications of Sunaga and Warmus, Classical interval arithmetic or set-theoretical interval arithmetic was greatly popularized by Ramon E.Moore in 1966 [2]. Classical interval arithmetic defines all the mathematical operations under intervals with their bounds of real numbers.

The first reason behind using classical interval arithmetic is to control round off errors resulting from using floating point computations and to put bounds on measurement errors in mathematical computations [2], [4]. After that, classical intervals analysis was found to be a good tool in so many branches of applied mathematics like solving linear and non linear equations, differential equations and global optimization [3], [4].

1.2.2 Basic Idea

The basic idea behind classical intervals is that instead of working with an uncertain real value (x) we can work with the two bounds of the interval $[a, b]$ which contains x (x lies between a and b) or could be one of the bounds. Consider for example, instead of estimating the height of someone using standard arithmetic as 2.0 meters, using interval arithmetic we might be certain that the person is somewhere between 1.97 and 2.03 meters [2], [28]. In other words, Classical interval arithmetic states the range of possible

outcomes explicitly. Simply, results are no longer stated as numbers, but as intervals which represent imprecise values. The sizes of the intervals are similar to error bars to a metric in expressing the extent of uncertainty. Simple arithmetic operations, such as basic arithmetic and trigonometric functions, enable the calculation of outer limits of intervals [2], [28].

Due to the heavy usage of classical interval analysis in so many mathematical applications, it found its way in many scientific and engineering fields. Some of these fields are chemical engineering, computer graphics, computer aided design tools, electrical engineering, Robotics, Control and so many other fields [2], [3], [4].

1.2.3 Definition

The definition of classical intervals is as follows [2]:

$$I(R) = \{[a,b] \mid a,b \in R, a \le b\}$$

1.2.4 Interval Extensions of Continuous Functions

One of the main objectives of classical interval analysis is to obtain an optimal interval extension in $I(\boldsymbol{R})$ for each real function in $\boldsymbol{R}$

1.2.4.1 Interval Extensions of Basic Operations

The basic operations of interval arithmetic are, for two intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$ that are subsets of the real line (-∞,∞):

$$A + B = [a_1 + b_1, a_2 + b_2]$$
$$A - B = [a_1 - b_2, a_2 - b_1]$$
$$A * B = [\min(a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2), \max(a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2)]$$
$$A / B = [\min(a_1 / b_1, a_1 / b_2, a_2 / b_1, a_2 / b_2), \max(a_1 / b_1, a_1 / b_2, a_2 / b_1, a_2 / b_2)], 0 \notin B$$

The addition and multiplication operations have the following properties [2], [28]:

Commutative

$$X \times Y = Y \times X$$

$$X + Y = Y + X$$

Associative

$$X \times (Y \times Z) = (Y \times X) \times Z$$

$$X + (Y + Z) = (Y + X) + Z$$

Sub-distributive

$$X \times (Y + Z) \subseteq X \times Y + X \times Z$$

As stated above, the interval multiplication operation (or interval division operation) costs four real multiplication operations and six comparisons (three comparisons for each bound of the resulting interval). There are other formulas which reduce the number of multiplication or division operations using signs tests on input intervals bounds as following:

$$A * B = \begin{cases} if\ a_1 \geq 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_1, a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0\ then\ [a_2 b_1, a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_1, a_1 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_2, a_2 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0 \\ \qquad then\ [min(a_2 b_1, a_1 b_2), max(a_1 b_1, a_2 b_2)] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_1, a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_2, a_2 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 \geq 0\ then\ [a_1 b_2, a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_2, a_1 b_1] \end{cases}$$

$$A/B = \begin{cases} if\ a_1 \geq 0, a_2 \geq 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_2, a_2/b_1] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_2, a_1/b_1] \\ if\ a_1 < 0, a_2 \geq 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_1, a_2/b_1] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_2, a_1/b_2] \\ if\ a_1 < 0, a_2 < 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_1, a_2/b_2] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_1, a_1/b_2] \end{cases}$$

Except for case 3 in multiplication we have only two multiplication or division operations and sign tests instead of four multiplication or division operation and six comparisons. This dramatically affects the performance of interval multiplication and division algorithms either in software or hardware implementations as we will see for the multiplication operation.

1.2.4.2 Interval Extensions of Monotonic Functions

Interval methods can also apply to functions which do not just use simple arithmetic such as functions that have monotonicity properties [2], [28].

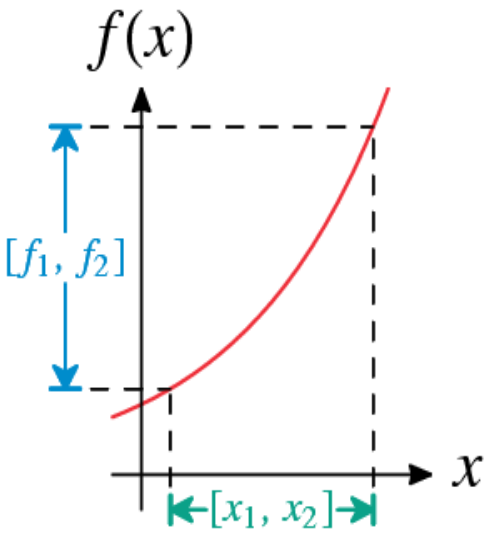

Figure 1.1: Increasingly Monotonic Function

If $f: \mathbb{R} \to \mathbb{R}$ is monotonically rising or falling in the interval$[x_1, x_2]$, then for all values in the interval $y_1, y_2 \in [x_1, x_2]$ such that $y_1 \leq y_2$, one of the following inequalities applies:

$$f(y_1) \leq f(y_2) or f(y_1) \geq f(y_2)$$

Thus the range corresponding to the interval $[y_1, y_2] \subseteq [x_1, x_2]$ can be calculated by applying the function to the endpoints y_1 and y_2 [2], [28]:

$$f([y_1, y_2]) = [\min\{f(y_1), f(y_2)\}, \max\{f(y_1), f(y_2)\}]$$

More generally, one can say that for piecewise monotonic functions it is sufficient to consider the endpoints x_1, x_2 of the interval $[x_1, x_2]$, together with the so-called critical points within the interval (those points where the monotonicity of the function changes direction) [2], [28].

1.2.4.3 Interval Extensions of Elementary Functions

We can easily deduce formulas to calculate the interval results for elementary functions (due to the monotinicity properties) as following [2]:

Exponential function:

$$a^{[x_1,x_2]} = [a^{x_1}, a^{x_2}], \text{for } a > 1$$

Logarithmic function:

$$log_a([x_1, x_2]) = [log_a x_1, log_a x_2], for\ positive\ intervals\ [x_1, x_2] and\ a > 1$$

Power function:

For Odd n ∈ N

$$[x_1, x_2]^n = [x_1^n, x_2^n]$$

For Even n ∈ N

$$[x_1, x_2]^n = [x_1^n, x_2^n], if x_1 \geq 0$$
$$[x_1, x_2]^n = [x_2^n, x_1^n], if x_2 < 0$$

$$[x_1, x_2]^n = [0, \max(\{x_1^n, x_2^n\})], otherwise$$

For even powers, the range of values being considered is important, and needs to be dealt with before doing any multiplication [2], [28].

For the sine and cosine functions, the critical points are at $(1/2 + n).\pi$ or $.\pi$ for all $n \in \mathrm{Z}$ respectively. Only up to five points matter as the resulting interval will be $[-1,1]$ if at least half a period is in the input interval. For sine and cosine, only the endpoints need full evaluation as the critical points lead to easily pre-calculated values namely -1, 0 , +1 [2], [28].

1.2.4.4 Interval extensions of General Functions

In general, we can combine the function rules $f(x_1, x_2, \dots\dots\dots x_n)$ with the equivalents of the basic arithmetic and elementary functions. This is called natural interval extension. We should note that there may be more than one real expression that are equivalent but each one has interval extension that is not equivalent to the other [2]. For example:

$$F(X) = X(1 - X)$$
$$G(X) = X - X^2$$
$$H(X) = 1/4 - (X - 1/2)^2$$

$F(X), G(X), H(X)$ are the interval extensions of the corresponding real functions $f(x), g(x), h(x)$ (capital letters denotes the interval extensions and small letters denotes the real functions).we should note that $f(x), g(x), h(x)$ are equivalent as a real functions but their interval extensions are not equivalent and each has different interval result (this is one of the problems of the classical interval analysis as we will discuss in the next section). Consider for example the interval results for $X = [0,1]$.

$$\begin{aligned} F([0,1]) &= [0,1] \times ([1,1] - [0,1]) \\ &= [0,1] \times [0,1] \\ &= [0,1] \end{aligned}$$

$$G([0,1]) = [0,1] - [0,1]^2$$

$$= [0,1] - [0,1]$$
$$= [-1,1]$$

$$H([0,1]) = 1/4 - ([0,1] - [1/2\,, 1/2])^2$$
$$= 1/4 - [1/2\,, 1/2]^2$$
$$= [1/4\,, 1/4] - [0, 1/4]$$
$$= [0, 1/4]$$

One of the important points of interval analysis is to obtain the interval extension that gives the tightest result.

1.2.5 Problems in Classical Intervals

Unfortunately, there are problems that appeared in classical interval analysis. Some of them are mentioned below [5] , [6].

1- Amplification of Dependence
When we calculate interval functions, we may have wider interval result than the actual result. Consider for example

$$f(x) = x - x\ \{x - x | x \in [1,2] = [0,0]\}$$

While with interval operation on $I(R)$

$$F(X) = X - X = [1,2] - [1,2] = [-1,1]$$

2- Sub-Distributive Law
The distributive property of multiplication is weakened in interval multiplication

$$A.(B + C) \subseteq A.B + A.C$$

For example:

$$[1,3]([1,1] + [-1,-1]) = [0,0]$$
$$while\ [1,3].[1,1] + [1,3].[-1,-1]\ = [-2,2]$$

$$clearly, [0,0] \subseteq [-2,2]$$

3- No Additive Inverse

In the real space $-x$ is the additive inverse of x such that $x - x = 0$ but in the interval space $X - X \neq [0,0]$

$$[2,5] - [2,5] = [-3,3]$$

4- No Multiplicative Inverse

In the real space $1/x$ is the multiplicative inverse of x such that $x.\left(\frac{1}{x}\right) = 1$ but in the interval space $X/X \neq [1,1]$

$$[2,5]/[2,5] = [2,5].[1/5\,,1/2] = [2/5\,,5/2]$$

5- Failure to solve some interval equations

Some interval equations couldn't be solved in the I(R) space. Even the simple equations like $[a,b] + [x,y] = [0,0]$. If [a, b] is a non degenerate interval (a < b), there exists no interval $[x,y]$ solves this simple equation. Also the linear equation $[a,b] + [x,y] = [c,d]$ the $I(R)$ system fails to obtain the solution from any set-theoretical interval operation between $[a,b]$and $[c,d]$. For Example

$$[1,3] + [x,y] = [4,5]$$
$$[x,y] = [4,5] - [1,3] = [1,4]$$

the solution is $[x,y] = [1,4]$ then if we substitute in the equation we have $[1,3] + [1,4] = [2,7] \neq [4,5]$ but $[2,7] \supset [4,5]$.

Due to the problems that appeared in classical interval analysis; many extensions were proposed to solve these problems. One of them is the modal interval analysis which is believed to be the general case of classical interval arithmetic like complex numbers are the general case of real numbers [5] , [6].

1.3 Modal Intervals

1.3.1 Historical Background

As we mentioned before, the papers of T.Sunaga and M.Warmus are considered the first publications that talk about interval calculus in general [32], [33], [34]. Moreover, Sunaga and Warmus introduced some basic principles in the Modal Intervals. Sunaga proposed [1,3] as a solution of the equation

$$[1,2] + X = [2,5]$$

This solution can only be obtained by modal interval arithmetic (as we will see later).

$$X = [2,5] - Dual([1,2]) = [2,5] - [2,1] = [1,3]$$

The Dual operator that is mentioned above reverses the interval bounds (it will be discussed in more details latter). Also Warmus proposed (in paper [33]) the system:

$$I^*(\boldsymbol{R}) = \{[a,b] | a,b \in \boldsymbol{R}\}$$

which (as we will see later) is the basic definition of the modal intervals space. Formal algebraic properties of proper intervals (intervals that have $a \leq b$ or simply classical intervals) and improper intervals (intervals that have $a \geq b$) were studied by H.J Ortolf (1968) and by E. Kausher (1973)[7]. Modal intervals in its form now was conceived by E.Gardenes (1985) who put the grounding construction of modal interval analysis [6].

1.3.2 Aim of Modal Intervals

The aim of modal intervals is to have interpretation of the interval results. Consider for example if a cable reel has an actual length within A = [9, 11] m and another within B = [19, 21] m, by connecting them it would be possible to reach a length of A + B = [9, 11] + [19, 21] = [28, 32]. Is it possible to cover

any length between 28 and 32 m? The answer is no, since we cannot reach a length of 32 m if the actual lengths for the reels were 10 and 20 m. What has happened? We are confusing two different propositional headings:

$\exists x \in [28,32]$: there exists an element of [28, 32],

$\forall x \in [28,32]$: every element of [28, 32]

These are two opposed selection methods for values within an interval cannot be accounted for in classical intervals $I(\boldsymbol{R})$ [5].

The problem of tolerance modelling is another real world example that shows us why we need an interval system that accounts for interpreting the equations and their results as mentioned in [39]. Suppose that we need to model the tolerance values for certain industrial parts A, B and C that need to be assembled as shown in Figure 1.2(a). As shown in the figure we need to fulfil the equation $a + b = c$ for dimensions a, b and c in Figure 1.2(a). When we model this equation we need to model also the possibility of having uncontrollable parts (e.g. parts are given from another factory) that have uncontrollable tolerances. For example we may have parts A and B are given from another factory with a given tolerances for the dimensions a ,b as shown in Figure 1.2(b). In that case we need part C to fit A and B and this can be modelled with the interpretation:

$$\forall a \in A', \forall b \in B', \exists c \in C', a + b = c$$

From now, we will represent the classical interval with a single quote as in A', B' and C' (As we will see in the next sections modal intervals consists of one of the modal logics $\forall, \exists$ combined with classical intervals but don't stuck into this point for now)

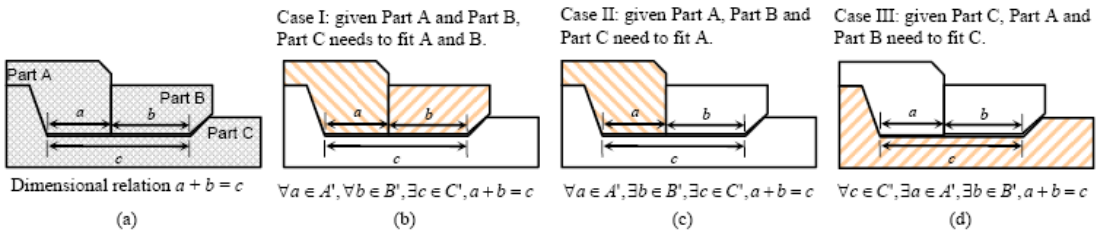

Figure 1.2: Different types of interpretations for tolerance modeling

In another case, we are given parts B and C as shown in Figure 1.2(c). Thus we need to fit parts B and C with A. This will be modelled with the interpretation:

$$\forall a \in A^{'}, \exists b \in B^{'}, \exists c \in C^{'}, a + b = c$$

Another case shown in Figure 1.2(d) which will be interpreted as:

$$\forall c \in C^{'}, \exists a \in A^{'}, \exists b \in B^{'}, a + b = c$$

The classical interval arithmetic cannot account for these different interpretations. On the other hand modal intervals are established to involve these different interpretations into arithmetic operations as we will see

1.3.3 Predicates and Quantifiers

Predicates and quantifiers are the foundation of modal theory. Together, they form the essential mathematical engine used to define the modal interval solution sets of real expressions [6].

An example of a predicate:

- P(x): x is greater than 3
  - P(x) is the statement
  - P is the propositional function
  - x is the subject
  - "is greater than 3" is the preditcate (a property the subject can have)

The purpose of the predicate is to transform the subject of the statement into a standard of truth, i.e., true or false. For this reason, the propositional function can be thought of as a Boolean function of one or more variables (subjects like x) [6]. The followings are some examples:

Example-1:

$$P(x): x > 3$$
$$P(4) = true$$
$$P(2) = false$$

Example-2:

$$Q(x, y): x = y + 3$$
$$Q(1,2) = false$$
$$Q(3,0) = true$$

Quantifiers "quantify" the truth of a statement by providing a mode of selection for a given variable in the predicate. In modal intervals there are exactly two modes to choose from, namely, ∀ (universal) and ∃ (existential). The ∀ and ∃ symbols are read "for all" and "there exists," respectively [6].

Given a statement P(x) and x ∈ D where x is a variable and D is a domain of values that x may take on, the proposition

$$(\forall x \in D)P(x)$$

requires P(x) to be true "for all" values in the domain of x while

$$(\exists x \in D)P(x)$$

requires P(x) to be true for at least one value in the domain of x, i.e., "there exists" in the domain of x an element such that P(x) is true [6].

1.3.4 Modal Intervals Building Blocks

The building blocks of the modal interval theory are:

- The set of real numbers R
- The set of classical intervals I(R)
- The set of classic predicates on the real line, $P(.): R \to \{true, false\}$

More particularly, if

$$Pred(R) \coloneqq \{P(.)|P(.): R \to \{true, false\}\}$$

is the set of classic predicates on the real line and

$$Pred(x) \coloneqq \{P(.) \in Pred(R)|P(x) = true\}$$

is the set of predicates a real number x accepts, then modal analysis stands on the identification

$$x \leftrightarrow Pred(x)$$

This is the main point of departure from the classical analysis which instead builds on a singleton interpretation of real numbers $x \leftrightarrow \{x\}$ [6].

A modal interval X is an element of the Cartesian product (X', Q) where X' is a classical interval and $Q \in \{\forall, \exists\}$ is one of the classic quantifier modes. To distinguish between modal interval and classical interval a prime symbol is put on the classical interval (as in X'). From this perspective, modal interval space can be defined as

$$I^*(R) \coloneqq \{(X', Q)|X' \in I(R), Q \in \{\forall, \exists\}\}$$

This is a similar method to that in which real numbers are associated in pairs having the same absolute value but opposite signs. Modal intervals in the system $I^*(R)$ are likewise associated in pairs having the same set but opposite modes [6].

Modal Intervals can be defined using another notation. Let $a, b \in \boldsymbol{R}$ then the canonical notation of a modal interval is:

$$[a,b] = \begin{cases} ([\mathrm{a},\mathrm{b}]', \exists) \text{ if } \mathrm{a} \leq \mathrm{b} \\ ([\mathrm{b},\mathrm{a}]', \forall) \text{ if } \mathrm{a} \geq \mathrm{b} \end{cases}$$

With the canonical notation, another definition for the set of modal interval is as following:

$$I^*(\boldsymbol{R}) = \{[a,b] | a,b \in \boldsymbol{R}\}$$

This reveals another reason why modal intervals are an extension of the classical intervals. In words, $\mathrm{I}(\mathrm{R})$ is isomorphic to a portion of $\mathrm{I}^*(\mathrm{R})$, namely the existential modal intervals [5],[6].

The importance of canonical notation comes from that all the mathematical properties of modal intervals are derived from this notation [5], [6], [29].

Some properties of a modal interval $\mathrm{X} \coloneqq [\mathrm{a},\mathrm{b}]$ are as following:

$$\mathrm{Inf}(\mathrm{X}) \coloneqq \mathrm{a}$$

$$\mathrm{Sup}(\mathrm{X}) \coloneqq \mathrm{b}$$

$$\mathrm{Mode}(\mathrm{X}) \coloneqq \begin{cases} \exists \text{ if } \mathrm{a} \leq \mathrm{b} \\ \forall \text{ if } \mathrm{a} \geq \mathrm{b} \end{cases}$$

$$\mathrm{Set}(\mathrm{X}) \coloneqq [\min(\mathrm{a},\mathrm{b}), \max(\mathrm{a},\mathrm{b})]'$$

The derivation of the canonical notation and its properties can be found in [29]. $\mathrm{Inf}(\mathrm{X})$ and $\mathrm{Sup}(\mathrm{X})$ are important definitions which form the canonical coordinates of a quite important diagram called (Inf, Sup)-diagram as shown in Figure 1.3.

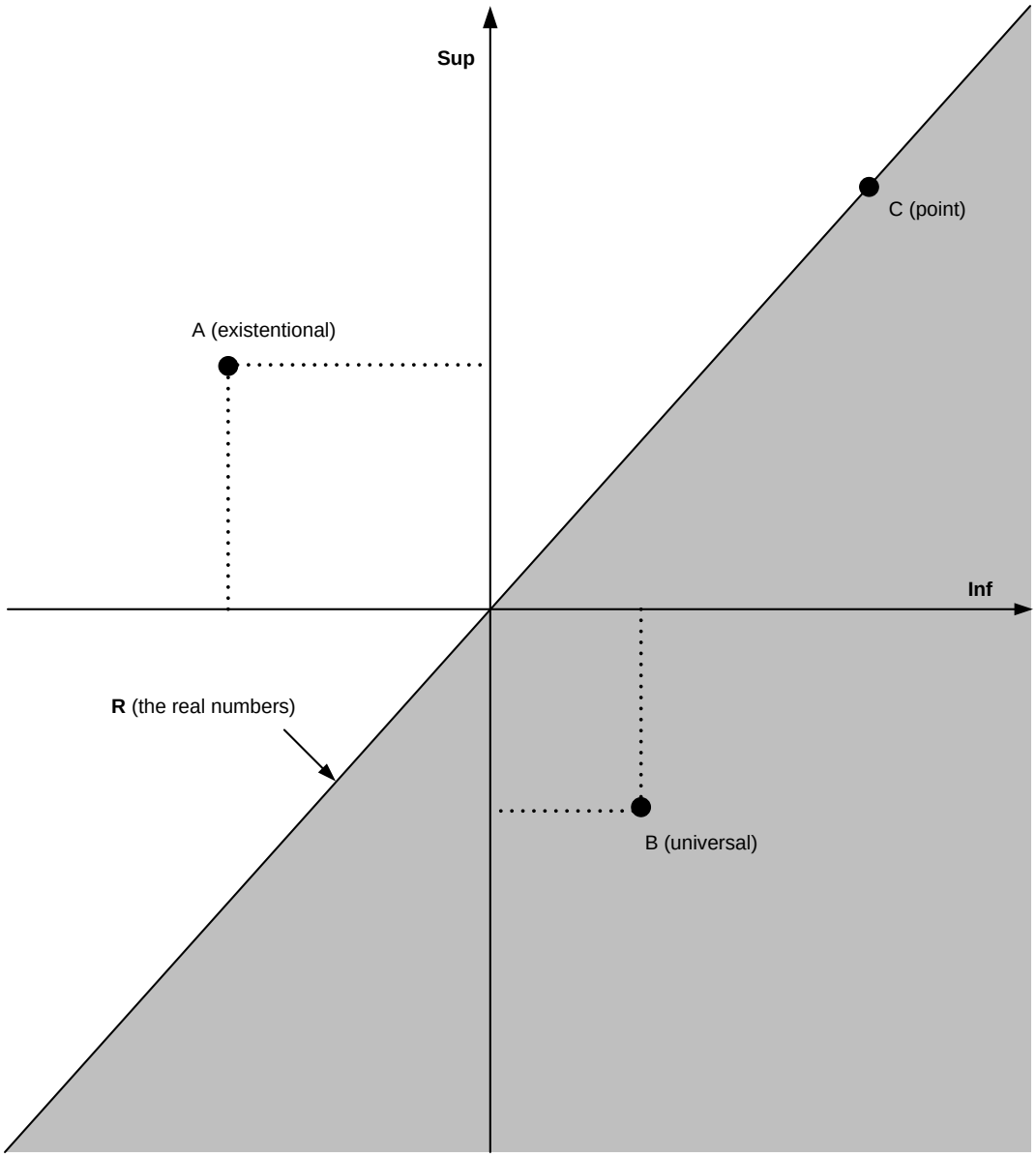

Figure 1.3: (Inf, Sup)-Diagram

This Diagram is useful as it reveals the underlying structure of the modal intervals. The (Inf = Sup) line is the set of all real numbers, i.e., the set of degenerate modal intervals. The half plane above is the set of existential intervals, and the half plane below is the set of universal modal intervals. For degenerate modal intervals, quantifier modes “for all” and “there exists” coincide, i.e., they have the same meaning. The (Inf, Sup)-diagram reveals the structural difference between the classical and modal intervals. For example, the shaded area below the (Inf = Sup) line represents a set of invalid intervals

that do not belong to the $I(R)$ system. But this is the set of universal modal intervals in the $I^*(R)$ system. If one views the (Inf, Sup)-diagram as an interval analogy of R divided into complimentary sets of positive and negative real numbers, a geometric insight is then provided into why $I(R)$ is not structurally complete. Restricting interval arithmetic to $I(R)$ is, by analogy, like restricting real arithmetic on R to the non-negative real numbers. Only the system $I^*(R)$ completes the analogy by providing complementary sets of intervals which are the existential and universal modal intervals [6].

Other names for the existential and universal intervals are the proper and improper intervals respectively. This comes from that for existential (proper) intervals $a \leq b$ while the universal (improper) intervals $a \geq b$.

1.3.5 Interval Extensions of Continuous Functions

1.3.5.1 Modal Semantic Extensions

One of the basic objectives of the interval analysis is to find an interval extension to any real function in **R**. In the Modal Interval Analysis, we have two objectives:

1- Find an interval extension R^n to R for a given real function f from $I^*(\mathbf{R}^n)$ to $I^*(\mathbf{R})$
2- The semantic meaning of the interval extension of that function

The semantic meaning of the interval extensions is one of the major advantages that the modal intervals have over the classical intervals. To obtain a semantic meaning to the interval computations, we must relate the modal interval extensions to one of the two semantic extensions (* and ** extensions) which play the grounding role in the modal intervals theory by providing the semantic meaning (interpretations) to the interval calculations [5], [30].

The definitions of the two semantic extensions are as following:

Given a real function f is an $\mathbf{R}^n$ to $\mathbf{R}$ continuous function, $X = (X_p, X_i) \in I^*(\mathbf{R}^n)$ (where X is a given vector of modal interval variables which can be divided into a vector of variables that have improper interval values X_i and a vector of variables that have proper interval values X_p), $x_p \in X'_p, x_i \in X'_i$ (where X'_p, X'_i are the classical counterparts of X_p, X_i)then:

$$f^*(X) :=$$
$$= \left[\min(x_p, X'_p) \max (x_i, X'_i) f(x_p, x_i), \max(x_p, X'_p) \min(x_i, X'_i) f(x_p, x_i)\right]$$

$$f^{**}(X) :=$$
$$= \left[\max(x_p, X'_p) \min (x_i, X'_i) f(x_p, x_i), \min(x_p, X'_p) \max(x_i, X'_i) f(x_p, x_i)\right]$$

We should note that in case there is no improper interval i.e. $X_i = \emptyset$ then:

$$f^*(X) = f^{**}(X) = \left[\min(x_p, X'_p) f(x_p), \max(x_p, X'_p) f(x_p)\right]$$

Which is the united extension of the real continuous function in classical interval analysis [2], [5].The derivations of the previous semantic extensions can be found in [30].

The semantic extensions f^* and f^{**} can be equal or not, but (unfortunately) both f^* and f^{**} are out of reach for any direct computation (This comes from applying the original definitions of f^* and f^{**} mentioned above), except for simple real functions such as basic arithmetic operations. Fortunately, we can relate the modal interval extensions of some specific functions to the * and ** semantic extensions such that we can obtain a meaningful result. Theorems 4.3 through 4.9 in [5] specify the rules that make a modal interval extension interpretable by one of the semantic extensions (f^* and f^{**}).

One remaining thing in the * and ** semantic extensions is the semantic meaning of the interval results. Theorem 4.1 and Theorem 4.2 in [5], [30] reveal completely the meaning of the interval results of f^* and f^{**}. The details of these theorems are stated in [5], [30].

1.3.6 Modal Interval Extensions of Basic Arithmetic Operations

The modal interval extensions of the basic interval operations (+, -, *, /) can follow f*-extension or f**-extension [5], [30], [31]:

If $A = [a_1, a_2]\ and\ B = [b_1, b_2]$ are modal intervals then

$$A \circledcirc B = f^*(A, B) = f^{**}(A, B)$$
$$\text{where } \circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$$

We should note that f*-extension and f**-extension are equal in the case of basic operations. This is mentioned in details in [5], [30], [31].

From the above we can conclude the following formulas for the basic modal interval arithmetic operations (addition, subtraction, multiplication and division) using the interval bounds [5], [30].

$$A + B = [a_1 + b_1, a_2 + b_2]$$

$$A - B = [a_1 - b_2, a_2 - b_1]$$

$$A * B = \begin{cases} if\ a_1 \geq 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_1, a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 \geq 0, b_2 < 0\ then\ [a_1 b_1, a_1 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0\ then\ [a_2 b_1, a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_1, a_1 b_2] \\ if\ a_1 \geq 0, a_2 < 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_1, a_2 b_1] \\ if\ a_1 \geq 0, a_2 < 0, b_1 \geq 0, b_2 < 0 \\ \quad then\ [max(a_1 b_1, a_2 b_2), min(a_2 b_1, a_1 b_2)] \\ if\ a_1 \geq 0, a_2 < 0, b_1 < 0, b_2 \geq 0\ then\ [0,0] \\ if\ a_1 \geq 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_2, a_1 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_2, a_2 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 \geq 0, b_2 < 0\ then\ [0,0] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0 \\ \quad then\ [min(a_2 b_1, a_1 b_2), max(a_1 b_1, a_2 b_2)] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_1, a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 \geq 0, b_2 \geq 0\ then\ [a_1 b_2, a_2 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 \geq 0, b_2 < 0\ then\ [a_2 b_2, a_2 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 \geq 0\ then\ [a_1 b_2, a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2 b_2, a_1 b_1] \end{cases}$$

$$A/B = \begin{cases} if\ a_1 \geq 0, a_2 \geq 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_2, a_2/b_1] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_2, a_1/b_1] \\ if\ a_1 \geq 0, a_2 < 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_2, a_2/b_2] \\ if\ a_1 \geq 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_1, a_1/b_1] \\ if\ a_1 < 0, a_2 \geq 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_1, a_2/b_1] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_2, a_1/b_2] \\ if\ a_1 < 0, a_2 < 0, b_1 > 0, b_2 > 0\ then\ [a_1/b_1, a_2/b_2] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [a_2/b_1, a_1/b_2] \end{cases}$$

We should note that:

1- For multiplication and division, we have different output depending on the input intervals bounds signs.

2- The modal interval basic operations formulas coincide with Kaucher interval basic operations[7] (Kaucher Intervals were proposed by E.Kaucher in 1980. They are another extension to the classical intervals which take into account the improper intervals in the interval

computations and try to solve the major shortages which we mentioned in the classical interval space. For further information about kaucher intervals, review [7]).

3- Classical basic operations formulas are subset of modal basic operations formulas which may indicate that modal intervals are the generalization form of classical intervals.

Another important definition is the Dual operator which is used in many modal theorems. One of its uses is to express inner rounding in terms of outer rounding as we will see later [5].

$$If\ A = [a_1, a_2]\ then\ Dual(A) = [a_2, a_1]$$

1.3.7 Advantages over classical intervals:

The modal intervals managed to solve some of the problems in classical intervals[5], [6].

1- There is additive inverse in modal intervals
   For example, the additive inverse of $X = [0,1]$ is $-Dual(X) = -Dual([0,1]) = -[1,0] = [0,-1]$ such that $X - Dual(X) = [0,1] - [1,0] = [0,1] + [0,-1] = [0,0]$

2- There is multiplicative inverse in modal intervals
   For example, the multiplicative inverse of $X = [1,2]$ is $1/Dual(X) = 1/[2,1]$ such that $X \times (1/Dual(X)) = [1,2] \times 1/[2,1] = [1,2] \times [1,1/2] = [1,1]$

3- The sub-distributive law is stronger than that of the classical

   $$Impr(A) \cdot B + A \cdot C \subseteq A \cdot (B + C) \subseteq Prop(A) \cdot B + A \cdot C$$

   Given that

   $$Prop([a,b]) = [\min(a,b), \max(a,b)]$$
   $$Improp([a,b]) = [\max(a,b), \min(a,b)]$$

For example,

$$[1,3] \cdot ([1,1] + [-1,-1])$$
$$= [3,1] \cdot [1,1] + [1,3] \cdot [-1,-1]$$
$$= [3,1] + [-3,-1] = [0,0]$$

While in the classical:

$$[1,3]' \cdot [1,1]' + [1,3]' \cdot [-1,-1]' = [-2,2]'$$

4- Not only solving the interval equations that the classical failed in solving it but also obtaining a meaningful interval results when solving these equations (As shown in the example in the "Aim of Modal Intervals" section)

Unfortunately, the dependency problem is not solved in the modal intervals but in some cases we can obtain tighter intervals than the classical ones (as in the case of the sub-distributive law).

To show the significant difference between the classical and modal intervals in real applications, let us consider the simulation of the derivative control process as stated in [39]. The derivative equation is as following:

$$\frac{dv}{dt} = k_d(v_0 - v) - \frac{1}{s}(v - v_a)$$

Where:

v : sensored tooling speed

v_0 : nominal control speed

k_d : action factor of control

v_a : sensor shift due to surroundings

s : sensitivity factor of sensor

The numerical version of the above equation is as following:

$$v(k+1) = v(k) + k_d\big(v_0 - v(k)\big) - \frac{1}{s}(v(k) - v_a)$$

Where k is the simulation time.

Now, consider that we want to implement the uncertainty in the parameters of this equation; we simply implement the parameters with

intervals and use the corresponding interval extensions. The naive classical extension of the simulation equation is:

$$V(k+1) = V(k) + K_d[V_0 - V(k)] - \frac{1}{S}[V(k) - V_a]$$

The modal extension of the simulation equation is:

$$V(k+1) = V(k) + K_d[V_0 - Dual(V(k))] - \frac{1}{S}[Dual(V(k)) - V_a]$$

Consider now the uncertain parameters take the following interval values:

$V_0 = [240\,, 241]$

$K_d = [0.004\,, 0.005]$

$V_a = [2\,, 3]$

$S = [1000\,, 1001]$

$V(0) = [3\,, 3]$

Figure 1.4 shows the modelling of the tooling speed (V) with time for the midpoints real function, classical and modal interval extensions.

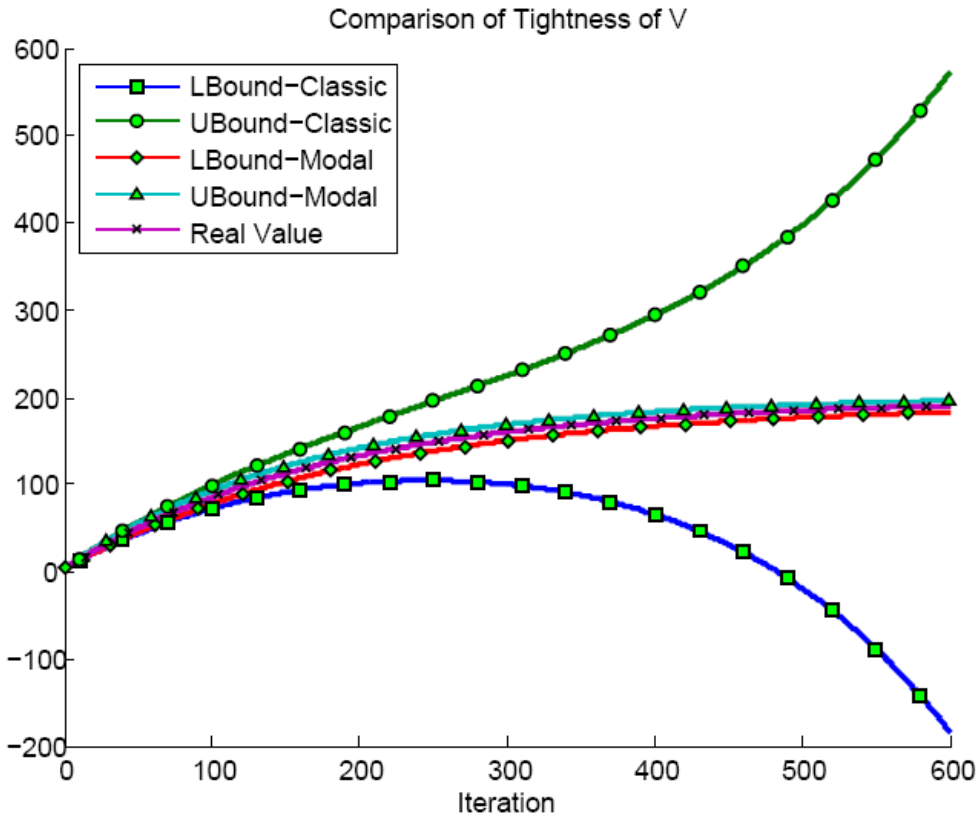

Figure 1.4: variation estimation based on modal interval compared to classic interval methods

The figure shows the significant difference between the modal and classical interval methods. It appears that the modal solution is tighter than the classical one. We should note also that the above naive classical extension not only has a much wider interval solution but also has totally wrong values (negative values) as shown in the figure.

Of course, the classical interval analysis provides many ways to overcome the amplification of wideness in the resulting intervals. One simple method is to reformulate the equation to decrease the severity of dependency problem [2]. Considering the previous Classical extension equation, if we rewrite it as following:

$$V(k+1) = V(k)[1 - K_d - 1/S] + K_d V_0 + V_a/S]$$

The classical interval results will be enhanced as seen in the Figure 1.5. Nevertheless, the modal results are tighter than that of the classical.

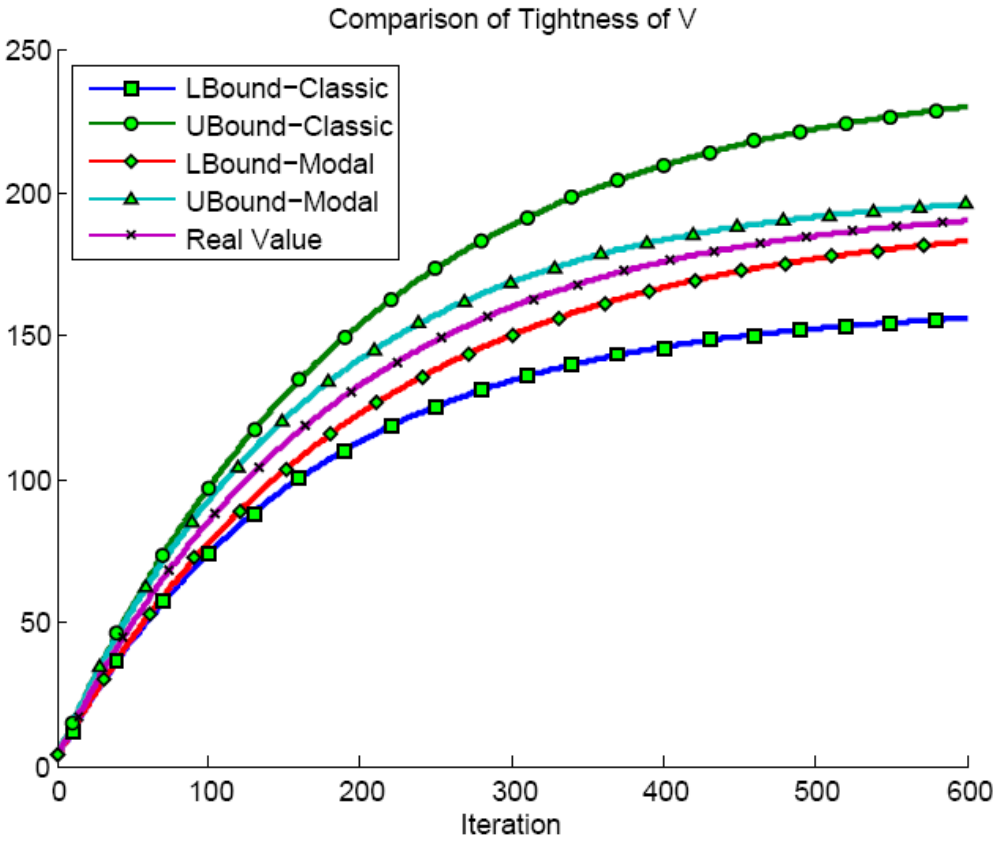

Figure 1.5: variation estimation based on modal interval compared to classic interval methods (equation reformulation)

Another method, to enhance the classical results, is to subdivide each input interval into N sub-intervals then recalculate the interval equation N-times. So we will have N-interval results which we make union for them to obtain the final result. As the subdivision method is not in the scope of this thesis, we will mention only the effect of using this method on the interval results (Check [2], [37] for more information about this method). Figure 1.6 shows the classical and modal interval results after uniformly subdividing each input interval (K_d, V_0, V_a, S) into N = 4 sub-intervals.

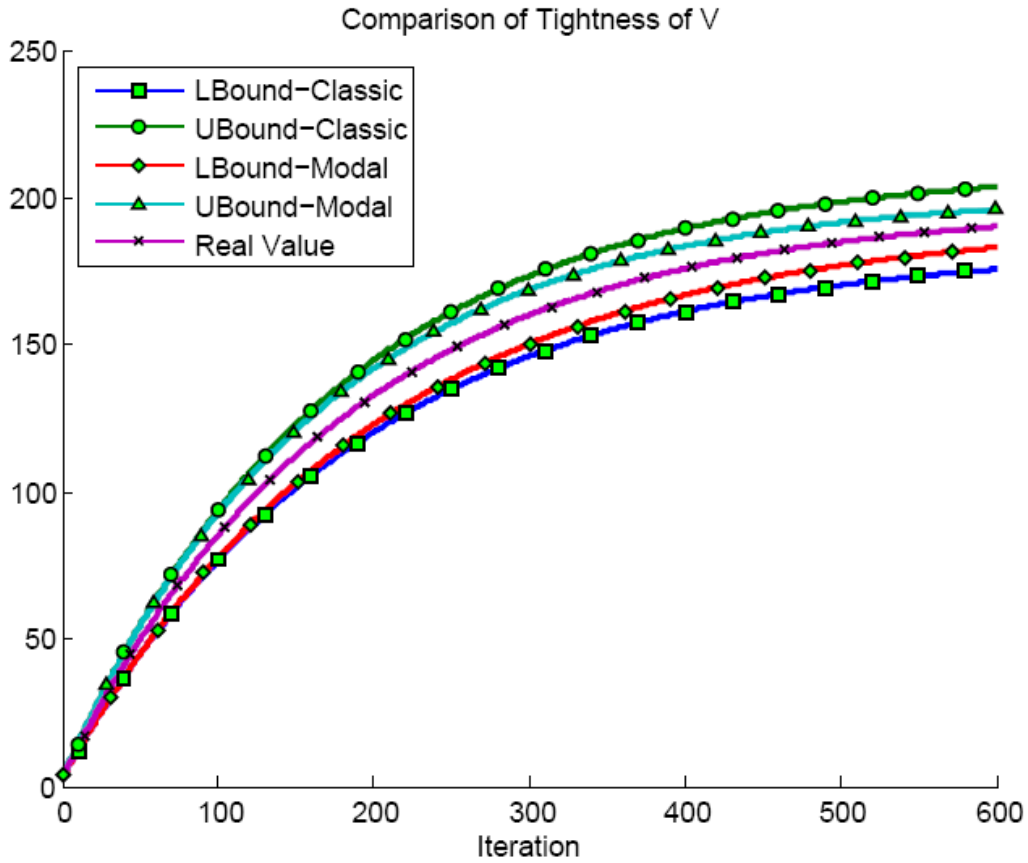

Figure 1.6: variation estimation based on modal interval
compared to classic interval methods (Sub-division method)

Clearly, the subdivision method enhances the classical results and makes them closer to the modal results. But this comes on the cost of consuming time as we repeat calculations N-times (as we have N Sub-intervals). Other examples that show the advantages of modal intervals over the classical

counterparts can be found in [5], [6], [39].

From the above discussion, we can see that the modal intervals space mathematically completes the classical intervals space and overcomes some of its problems. In chapter two, we will first introduce the concept of rounded interval arithmetic and how intervals can be implemented on digital systems (either in software or in hardware) then we will discuss the motivation behind this work. After that we will discuss some of the previous works done in that field. In chapter three, we will move to the hardware implementation of the Modal Interval Adder/Subtractor and Multiplier and show the results of two different implementations for each unit (serial and parallel implementations) on FPGAs and ASICs. In chapter four, we will discuss the testing results and show comparisons with other implementations. Also in chapter four, we will give a conclusion of the thesis work and the future work directions.

Chapter 2

2 Previous Work and Motivation

2.1 Rounded Interval Arithmetic

As we mentioned before in the Introduction section, the floating point representation of real numbers is not exact and due to this limitation, rounding errors occur in the numerical computations which results in catastrophic errors.

In case of representation of intervals, either modal or classical, on computers, we map the real interval bounds into floating point numbers with proper rounding for each bound. Unlike the floating point representation of the real numbers, the floating point representation of the two interval bounds with proper rounding for each bound leads to monitoring and controlling the errors in numerical computations.

In case of classical intervals, to correctly enclose all the interval values we make rounding out which means the lower interval bound rounded to -∞ (sometimes called rounded down) and the upper interval bound rounded to +∞ (sometimes called rounded up)[2].

$$Out([a_1, a_2]') = [\nabla a_1, \Delta a_2]'$$

When doing interval addition, subtraction, multiplication or division, the result of each operation is rounded down for lower bound and rounded up for the upper bound[8].

$$A'+B'=[\nabla(a_1+b_1),\Delta(a_2+b_2)]$$
$$A'-B'=[\nabla(a_1-b_2),\Delta(a_2-b_1)]$$
$$A'*B'=[\min(\nabla(a_1b_1),\nabla(a_1b_2),\nabla(a_2b_1),\nabla(a_2b_2)),\max(\Delta(a_1b_1),\Delta(a_1b_2),\Delta(a_2b_1),\Delta(a_2b_2))]$$
$$A'/B'=[\min(\nabla(a_1/b_1),\nabla(a_1/b_2),\nabla(a_2/b_1),\nabla(a_2/b_2)),\max(\Delta(a_1/b_1),\Delta(a_1/b_2),\Delta(a_2/b_1),\Delta(a_2/b_2))],0\notin B$$

Unfortunately, out rounding the interval is not always right if we talk about modal intervals. We may need inner rounding[5], [6]. Inner rounding is rounding the first interval bound up and the second interval bound down.

$$Inn([a_1,a_2])=[\Delta a_1,\nabla a_2]$$

To know why inner rounding is also needed, consider the following example:

The exact solution of the following equation:

$$[4/3\,,5/3]+[x_1,x_2]=[2,7]$$

$$\Rightarrow [x_1,x_2]=[2,7]-Dual([4/3\,,5/3])$$
$$=[2,7]+[-4/3\,,-5/3]$$
$$=[2/3\,,16/3]$$

But for $Out(A)+X=B$

$$Out([4/3\,,5/3])+[x_1,x_2]=[2,7]$$

$$\Rightarrow [x_1,x_2]=[2,7]-Dual([1.3,1.7])$$
$$=[2,7]+[-1.3,-1.7]$$
$$=[0.7,5.3]$$

Which doesn't contain the exact interval result, but for $Inn(A)+X=B$

$$Inn([4/3\,,5/3])+[x_1,x_2]=[2,7]$$

$$\Rightarrow [x_1,x_2]=[2,7]-Dual([1.4,1.6])$$
$$=[2,7]+[-1.4,-1.6]$$
$$=[0.6,5.4]$$

Which contains the exact result.

As we will see later, due to the Dual operator we can express the inner rounding in terms of the outer rounding so there is no need to implement inner rounding. Only outer rounding can be used.

2.2 Digital Representation

The two interval bounds can be digitally represented as fixed point or floating point numbers. Fixed point numbers are suitable for digital signal processing control units[9]. Floating point representation can be single, double or variable precision. As our main target is to add support for modal interval basic operations in common computer systems, the best solution is representing modal interval bounds using double precision binary floating point numbers. IEEE standard for double precision binary floating point numbers specifies 64 bit for each number as following[10]:

1 bit for the sign bit (S), 11 bits for the biased exponent (E) and 52 bits for the mantissa (M). The mantissa contains a hidden one according to the IEEE standard so the actual precision is 53 bits. The value of a normalized IEEE double precision number is

$$(-1)^{S} * 1.\mathrm{M} * 2^{\mathrm{E}-1023}$$

This means to minimally represent modal interval we need 128 bits each 64 bit representing one IEEE double precision binary floating point number.

2.3 Motivation

2.3.1 Modal versus classical intervals

Choosing to implement modal interval addition, subtraction and multiplication comes from the fact that modal intervals overcome some problems in classical intervals as we mentioned before. Besides, it is believed that modal intervals are the natural extension of classical intervals. Also modal interval analysis found the way to some applications in computer graphics and control[5], [6], [11].

2.3.2 Hardware versus Software Implementation

The problem of software implementation of intervals basic operations is the bad performance. Several reasons make software slower than hardware:

1- Changing rounding mode which causes a large overhead in case of pipeline architecture because of pipeline flushing.

2- Function's calls if interval operations are implemented as functions

3- Sign test and choosing the correct case which is in multiplication and division operations

Some software implementations may have all the reasons and others may have one or two. In general the need for hardware support for interval arithmetic is increasing due to the increase in applications using interval arithmetic and the need to have arithmetic units giving higher accuracy with a speed comparable to that of the normal floating point arithmetic units [3], [4], [6], [8].

This thesis presents the hardware implementation of the modal interval adder, subtractor and multiplier using IEEE double precision binary floating point adder, subtractor and multiplier.

2.4 Previous Work

2.4.1 Classical Intervals

Many papers describe the hardware implementation of classical interval basic operations. For example:

1- Hardware support for interval instructions is provided in UltraSPARC-III processors with the "Set Interval Arithmetic Mode" (SIAM) instructions. These instructions improve the efficiency of interval arithmetic by enabling the rounding mode bits in the floating-point status register (FSR) to be overridden without the resulting overhead of pipeline flush. It enables the interval rounding mode to be changed every cycle without flushing the pipeline. Typical interval performance improvement from the SIAM instruction has been measured to be approximately 30% [19], [26].

2- Changing in the architecture of the basic floating point arithmetic unit to overcome the problem of changing rounding mode and thus eliminates the pipeline flushing in case of interval arithmetic operations. As in [27], we can modify the double path adder architecture such that it works on two addition operations in the close and far path instead of one. The double path floating point adders are based on performing speculatively addition on two distinct low latency paths (CLOSE and FAR path) [35], [36]. The correct result is selected at the end of the computation. The modified addition unit exploits the parallelism of the double adder's structure by performing the two operations required for an interval addition/subtraction simultaneously, each on a different path. In order to do so, several changes have been made to the classic architecture of the floating point adder:
   a. The sign and exponents computation circuits have been duplicated (two signs and two exponents are computed).
   b. A dedicated module has been placed before the splitting of the two paths.

   The role is to dispatch the required operands (along with the effective operation and rounding mode) on their corresponding path. The two operations required for an interval addition/subtraction can be performed either simultaneously (favourable case), or sequentially. The

favourable case is the case when we have operations (effective additions and subtractions) with exponents' difference equal to 0 or 1 which can be executed properly on the CLOSE path.

Also the multiplier is modified such that it produces results with multiple rounding schemes (i.e. the multiplier has more than one result, one rounded towards negative infinity and the other rounded towards positive infinity). By this way we do not need to change rounding mode each cycle.

3- Fixed point Interval-ALU for digital signal processing and control applications [9]:

The overall architecture of the I-ALU can be seen in the block diagram shown in Figure 2.1. The hardware model is divided into four parts. the flag generator, lower bound and upper bound modules, and the rounding unit. The flag generator module is responsible for generating the control signals for the more complicated classical interval multiplication operation. Based on the signs of the input operands, the flag generator generates control signals to select the appropriate multiplication case among the nine cases that we mentioned earlier in classical intervals section. The lower bound module and the upper bound module calculate the lower and upper bounds of the output interval, respectively. These two modules are independent of each other and hence operate in parallel. The rounding unit implements the Outward Rounding of the interval result.

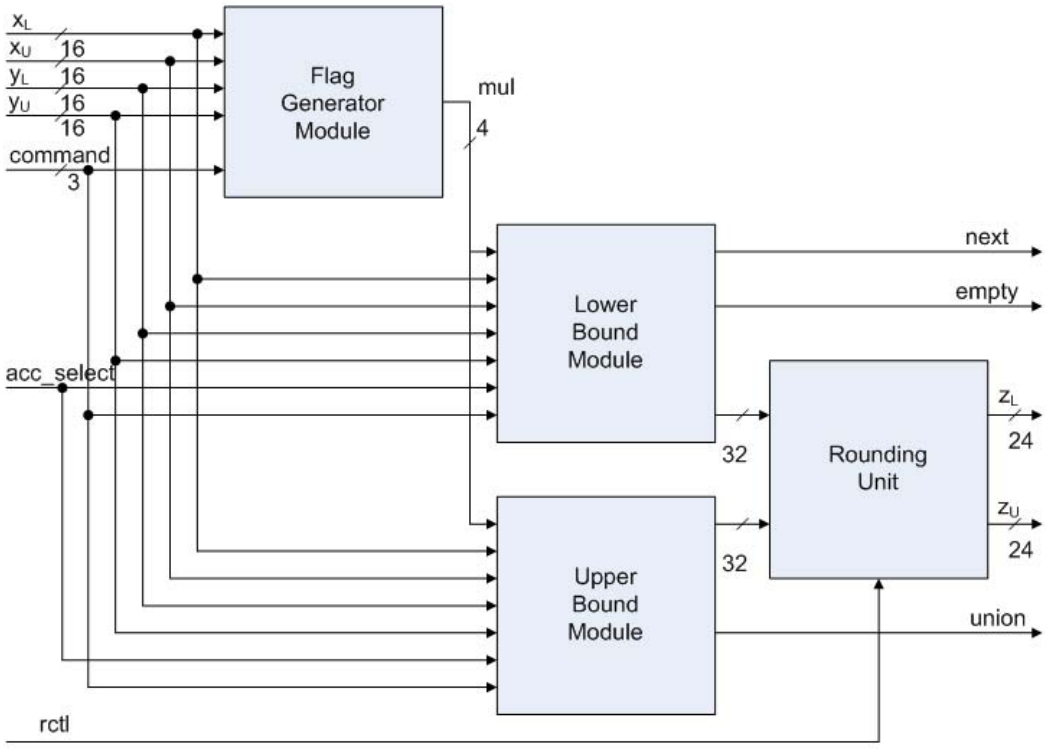

Figure 2.1: Block Diagram of I-ALU

The ALU is designed for operation on 16 bit input interval numbers in the two's complement form. The I-ALU operations are 16-bit fixed point interval addition, subtraction, multiplication and multiply-accumulate. This ALU is suitable for specific DSP and control applications which do not need high precision and have small range of variations to be suitable with fixed point implementation

4- A combined interval floating point arithmetic units that can work either in floating point mode (normal mode) or in interval mode [20],[21]:

These designs are based on the approach that an interval multiplier (or divider) can share hardware with an existing floating point multiplier (or divider), thereby achieving the performance benefits of a interval multiplier (or divider) at relatively low costs. The multiplier design does not solve the uncommon case of multiplication where both end-points contain zero. Instead, it resorts to software solutions to solve it. Interval multiplication (or division) in that case requires only one more cycle than floating point operation, and is one to two orders of magnitude faster than software implementations of interval multiplication.

5- Full hardware implementation for interval arithmetic operations (to be as fast as floating point operations) as suggested in [24]:

   The author suggests implementations for the interval basic operations (addition, subtraction, multiplication and division) such that the interval operations speed can be as fast as the normal floating point operations. This comes by using parallelism. To speed up interval operations, we should have two operation units (two adder, subtractors, multipliers or dividers). One unit is to calculate the result's lower bound and the other is to calculate the result's upper bound. Once again, we should notice that in case of multiplication and division the matters are a little more complicated as we reduce number of operations dependent on the input operands.

6- Variable precision interval arithmetic processors [22], [23]:

   The author presents designs, arithmetic algorithms and software support for a family of variable precision, interval arithmetic processors. These processors give the programmer the ability to detect, and if desired, to correct the implicit errors in finite precision numerical computations. The processors are two to three orders of magnitude faster than software packages that provide similar functionality.

2.4.2 Modal Intervals

There is no published work about the hardware implementation of the modal interval basic operations which is our point of research.

From the above, we can see that there are many contributions done in hardware support of classical interval basic operations while there is no work done for the modal interval hardware support. As we will see the cost of adding support to modal intervals may be lower than the cost of adding support to classical intervals only.

Chapter 3

3 Hardware Implementation

3.1 Modal Interval Double Floating Point Adder/Subtractor Implementation

Usually the addition and subtraction operations are combined into one unit thus the modal interval adder/subtractor will be one unit.

The definition of modal interval addition and subtraction with outer rounding as following:

If $A = [a_1, a_2]$, $B = [b_1, b_2]$ are modal intervals then

$$A + B = [\nabla(a_1 + b_1), \Delta(a_2 + b_2)]$$

$$A - B = [\nabla(a_1 - b_2), \Delta(a_2 - b_1)]$$

It should be noticed that from hardware point of view there is no difference between classical and modal intervals addition/subtraction except for

1- A, B are modal intervals

2- According to Theorems 4.5, 4.8 and 4.9 in [6], the DUAL operator may be used in addition or subtraction operation.

3- As mentioned before, outer rounding does not assure the enclosure of all results so inner rounding may be used.

The DUAL operator may be parsed and handled by the compiler without any overhead (No additional clock cycles to make the DUAL operation).

For inner rounding, from the following relations [5][6]:

$$Inn(X) = Dual(Out(Dual(X)))$$

$$Inn(A \circ B) = Dual(Out(Dual(A) \circ Dual(B)))$$

Where $\circ$ denotes addition or subtraction

We can express inner rounding in terms of outer rounding thus inner rounding may be realized also by the compiler.

From the above discussion, the modal interval double precision floating point adder/subtractor can be implemented exactly as the floating point classical interval adder/subtractor if we rely on the compiler to resolve the DUAL operator and inner rounding.

3.1.1 Handling Infinities in input intervals

One or the two bounds of the input operands may contain $\pm\infty$.This is called extended modal interval addition/subtraction. The following two tables write down all the cases of inputs including $\pm\infty$ for addition and subtraction respectively.

Addition	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty,+\infty)$	$(+\infty,-\infty)$	$(+\infty, b_2]$	$[b_1, -\infty)$
$(-\infty, a_2]$	$(-\infty, a_2+b_2]$	$(-\infty, a_2+b_2]$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	(Nan, $-\infty$)	(Nan, a_2+b_2]	$(-\infty,+\infty)$
$[a_1, a_2]$	$(-\infty, a_2+b_2]$	$[a_1+b_1, a_2+b_2]$	$[a_1+b_1,+\infty)$	$(-\infty,+\infty)$	$(+\infty,-\infty)$	$(+\infty, a_2+b_2]$	$[a_1+b_1, -\infty)$
$[a_1, +\infty)$	$(-\infty,+\infty)$	$[a_1+b_1,+\infty)$	$[a_1+b_1,+\infty)$	$(-\infty,+\infty)$	$(+\infty$, Nan)	$(+\infty,+\infty)$	$[a_1+b_1$, Nan)
$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	(Nan, Nan)	(Nan, $+\infty$)	$(-\infty$,Nan)
$(+\infty,-\infty)$	(Nan, $-\infty$)	$(+\infty,-\infty)$	$(+\infty$,Nan)	(Nan, Nan)	$(+\infty,-\infty)$	$(+\infty,-\infty)$	$(+\infty,-\infty)$
$(+\infty, a_2]$	(Nan, a_2+b_2]	$(+\infty, a_2+b_2]$	$(+\infty,+\infty)$	(Nan, $+\infty$)	$(+\infty,-\infty)$	$(+\infty, a_2+b_2]$	$(+\infty,-\infty)$
$[a_1, -\infty)$	$(-\infty,-\infty)$	$[a_1+b_1, -\infty)$	$[a_1+b_1$, Nan)	$(-\infty$,Nan)	$(+\infty,-\infty)$	$(+\infty,-\infty)$	$[a_1+b_1, -\infty)$

Table 3.1: Extended Modal Interval Addition

Subtraction	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty,+\infty)$	$(+\infty,-\infty)$	$(+\infty, b_2]$	$[b_1, -\infty)$
$(-\infty, a_2]$	$(-\infty,-\infty)$	$(-\infty, a_2- b_1]$	$(-\infty, a_2- b_1]$	$(-\infty,+\infty)$	(Nan, $-\infty$)	$(-\infty,-\infty)$	(Nan, $a_2- b_1$]
$[a_1, a_2]$	$[a_1- b_2, +\infty)$	$[a_1- b_2, a_2- b_1]$	$(-\infty, a_2- b_1]$	$(-\infty,+\infty)$	$(+\infty, -\infty)$	$[a_1- b_2, -\infty)$	$(+\infty, a_2- b_1]$
$[a_1, +\infty)$	$[a_1- b_2, +\infty)$	$[a_1- b_2, +\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(+\infty$,Nan)	$[a_1- b_2$, Nan)	$(+\infty,+\infty)$
$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	$(-\infty,+\infty)$	(Nan, Nan)	$(-\infty$,Nan)	(Nan,$+\infty$)
$(+\infty,-\infty)$	$(+\infty$,Nan)	$(+\infty,-\infty)$	(Nan, $-\infty$)	(Nan, Nan)	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$
$(+\infty, a_2]$	$(+\infty$,Nan)	$(+\infty, a_2- b_1]$	(Nan, $a_2- b_1$]	(Nan, $+\infty$)	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, a_2- b_1]$
$[a_1, -\infty)$	$[a_1- b_2$, Nan)	$[a_1- b_2, -\infty)$	$(-\infty,-\infty)$	$(-\infty$,Nan)	$(+\infty, -\infty)$	$[a_1- b_2, -\infty)$	$(+\infty, -\infty)$

Table 3.2: Extended Modal Interval Subtraction

Note that the shaded part coincides with the classical cases as in [8]. From modal interval addition and subtraction equations, we can say that the modal interval addition/subtraction consists of two normal floating point addition/subtraction operations rounded to -∞ and +∞ respectively. Thus handling infinities in the modal interval addition/subtraction coincide with handling infinities in the normal floating point addition/subtraction with the proper rounding. Table 3.3: Floating Point Addition and Table 3.4: Floating Point Subtraction show the floating point addition/subtraction including ±∞ in inputs according to the IEEE 754 standard [10].

Addition	−∞	B	+∞
−∞	−∞	−∞	Nan
A	−∞	A+B	+∞
+∞	Nan	+∞	+∞

Table 3.3: Floating Point Addition

Subtraction	−∞	B	+∞
−∞	Nan	−∞	−∞
A	+∞	A−B	−∞
+∞	+∞	+∞	Nan

Table 3.4: Floating Point Subtraction

3.1.2 Hardware Implementation

The implementation of modal interval double floating point adder/subtractor can be realized by many ways. Two designs are presented here, one design to maximize the speed and the other to minimize the area.

3.1.2.1 Serial Interval Adder/Subtractor

3.1.2.1.1 Hardware Architecture

The high level architecture of the MIBFP Add/Sub (Modal Interval Binary Floating Point Adder/Subtractor) as in Figure 3.1

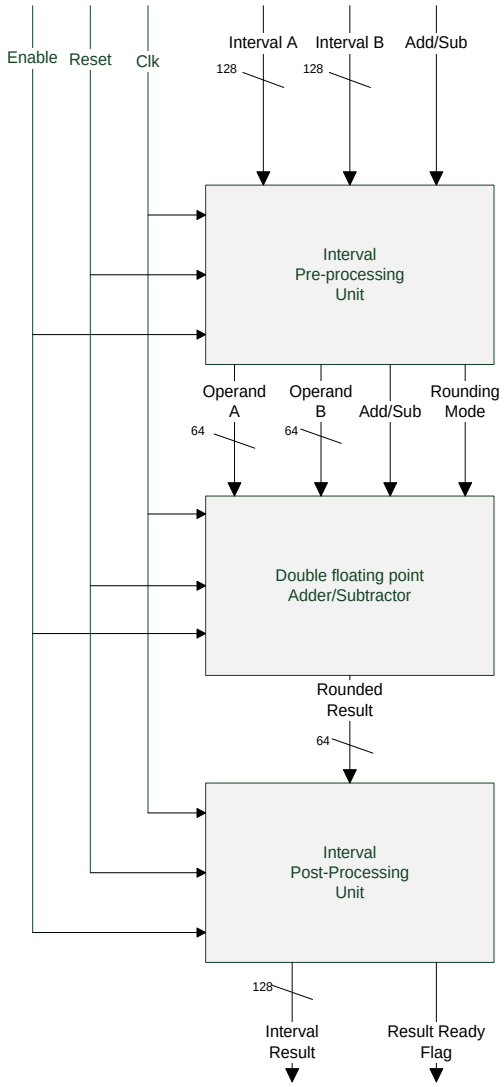

Figure 3.1: Modal Interval Double Floating Point Adder/Subtractor (Serial Implementation)

Interval Pre-processing unit:

Divides the interval operands into two sequential floating point addition/subtraction operations with the appropriate rounding mode for

each operation (the first operation which is the first bound in the interval result rounded to $-\infty$ and the other rounded to $+\infty$). The logic circuit of the pre-processing unit is shown in the following figure.

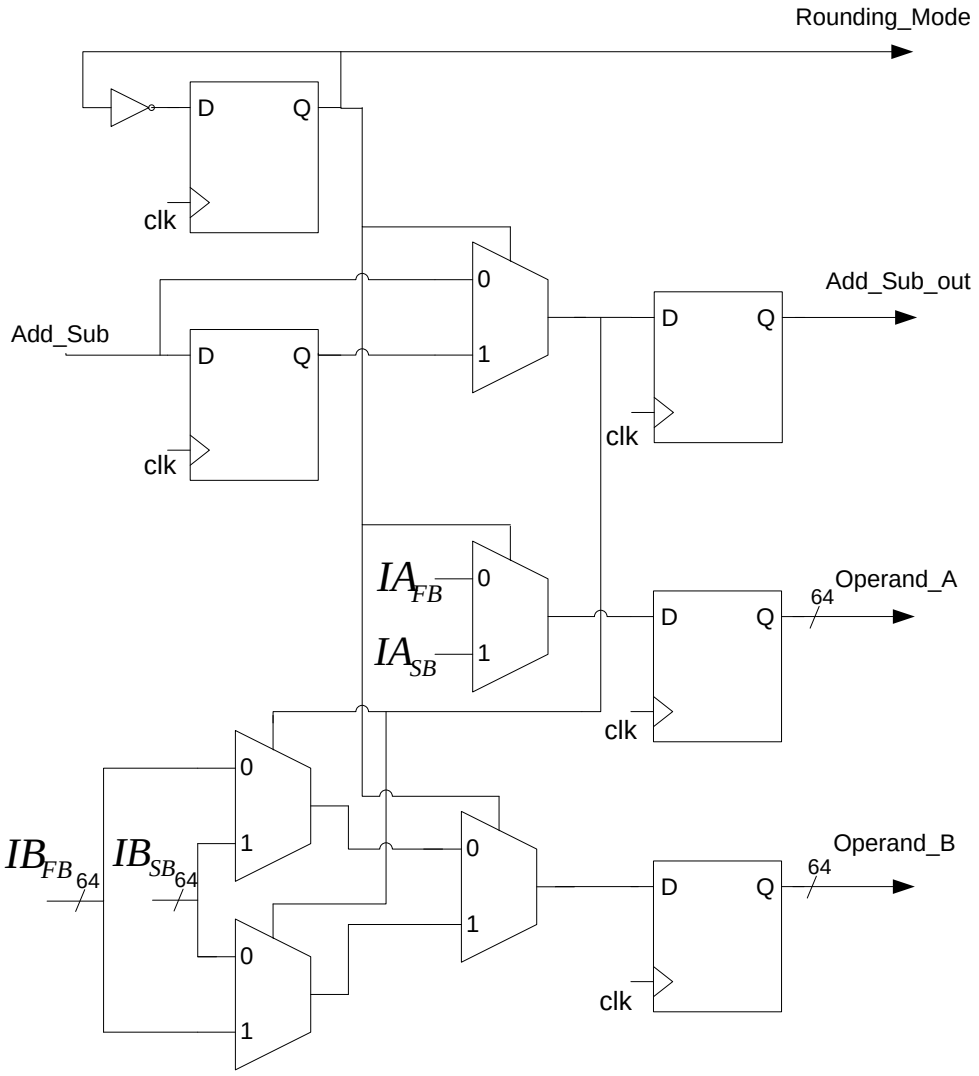

Figure 3.2: pre-processing unit logic circuit

BFP Adder/Subtractor:

It is a normal double floating point adder/subtractor. The floating point unit was originally built using the open source VHDL code of the IEEE-754 compliant double precision floating point core posted at opencores site [40]. The VHDL source code of the floating point adder and subtractor were massively modified to fix bugs in pipelining, handling special cases (like

denormalized numbers and infinities), reduces number of pipeline stages and to merge both units (adder and subtrtactor) into one Adder/Subtractor unit. The design and implementation of the floating point adder/subtractor unit will not be discussed as this is out of the scope of the thesis topic. However, the area and timings results of this unit are used to compare with the results obtained for the Modal interval designs.

Interval Post-Processing unit:

Collects the two floating point results into one interval result then raises a flag for ready result. The logic circuit of the post-processing unit is shown in the following figure.

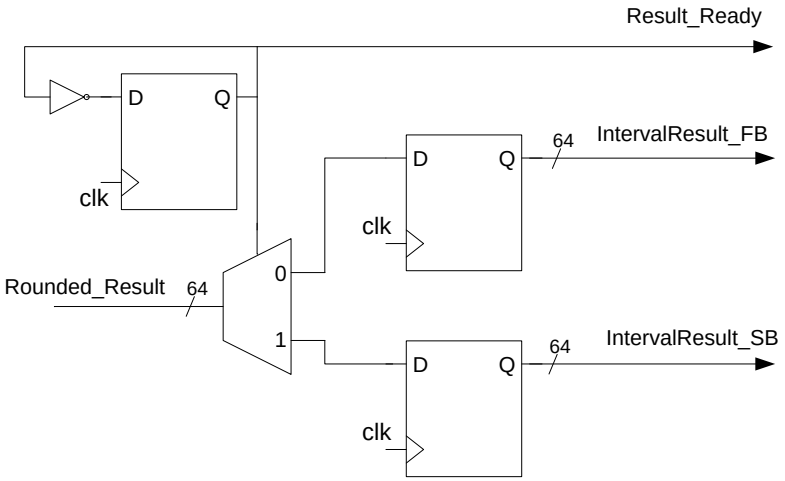

Figure 3.3: post-processing unit logic circuit

3.1.2.1.2 Pipeline stages

The following figure shows a schematic for the pipeline stages of executing two modal interval addition (or subtraction) operations.

Operation No.	Pipeline Stage											
1	Pre-Process C-1	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7	Post-Process C-1			
		Pre-Process C-2	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7	Post-Process C-2		
2			Pre-Process C-1	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7	Post-Process C-1	
				Pre-Process C-2	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7	Post-Process C-2
Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12

Figure 3.4: Pipeline stages for Serial MIBFP Adder/Subtractor

As shown, the pre-processing unit applies the inputs to the floating point adder/subtractor on two cycles (Pre-Process C-1 and Pre-Process C-2 cycles). The floating point adder/subtractor takes seven clock cycles to execute (from Adder/Sub C-1 to Adder/Sub C-7). The post-processing unit receives the first result's bound in the Post-Process C-1 cycle then receives the second result's bound in the Post-Process C-2 cycle and flags result ready.

3.1.2.1.3 Logic Synthesis

The modules are written in VHDL and synthesized using Quartus-II Altera software tool. Two types of Altera FPGAs are used to implement the MIBFP Adder/Subtractor as a prototype [12]:

1- Cyclone II (lower power, cost and speed)
Device EP2C35F672C6 of Cyclone II Family is used. The results are as shown in Table 3.5:

	Area		Timings		
	No. of LEs	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	1813	741	121	7	1
Serial MIBFP Adder/Subtractor	2007	950	120.5	10	0.5

Table 3.5: Area and Timings (Serial MIBFP adder/subtractor – Cyclone II)

2- Stratix III (higher power, cost and speed)
Device EP3SL50F780C2 of Stratix III Family is used. The results are as shown in Table 3.6:

	Area		Timings		
	No. of LEs	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	1178	745	250	7	1
Serial MIBFP Adder/Subtractor	1230	997	250	10	0.5

Table 3.6: Area and Timings (Serial MIBFP adder/subtractor – Stratix III)

The FPGA synthesis is good to make a fast prototype of the hardware implementation but it has the disadvantage of lower speed than ASIC speed. Besides, we don't have solid information about the area (in terms of area units not in terms of number of logic elements as in ALTERA FPGAs). So, the design is implemented using an ASIC standard cell library (the Nangate 45nm Open Cell Library) [13]. The synthesis is done using the Synopsis Design Compiler software tool. The results are as shown in Table 3.7:

	Area (mm2)			Timings		
	Combinational Area (mm^2)	Registers Area (mm^2)	Interconnect Area (mm^2)	Clock Frequency (GHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	0.0055	0.0039	0.0032	1.176	7	1
Serial MIBFP Adder/Subtractor	0.0059	0.0051	0.0037	1.176	10	0.5

Table 3.7: Area and Timings (Serial MIBFP adder/subtractor – Nangate 45nm)

As we notice from the previous tables that the pipeline throughput is 1 result per two clock cycles which is half the throughput of normal floating point adder/subtractor. Percentage increase in the area is about 16% of the normal BFP adder/subtractor.

The advantage of that design is that we use only one floating point adder/subtractor to implement modal interval adder/subtractor. This implementation decreases the area and power consumption. On the other hand it decreases the interval operation speed to half speed of the normal floating point operation which means that we have interval result each two clock cycles (in case of pipelining) as shown in tables 3.5, 3.6 and 3.7.

3.1.2.2 Parallel Interval Adder/Subtractor

3.1.2.2.1 Hardware Architecture

The high level architecture of the parallel BFP Modal Interval Adder/Subtractor is as in Figure 3.5

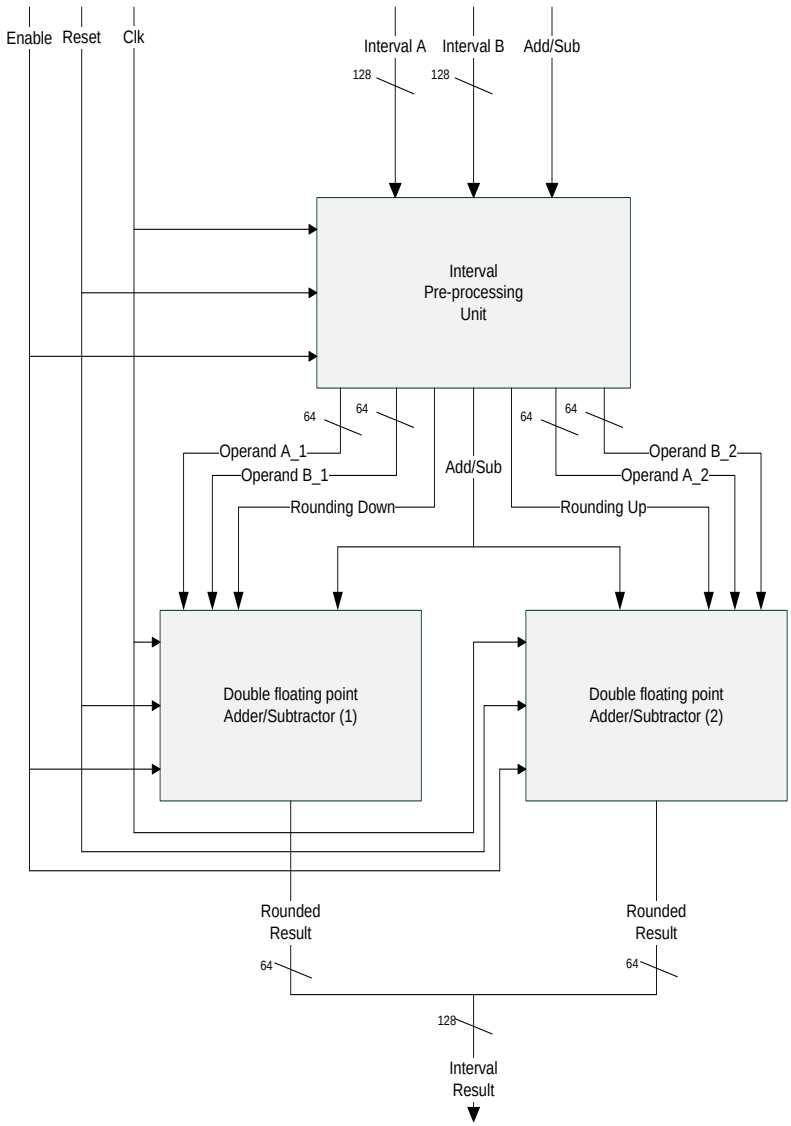

Figure 3.5: Modal Interval Double Floating Point Adder/Subtractor (Parallel Implementation)

Interval Pre-processing unit:

Divides the interval operands into two parallel floating point addition/subtraction operations with the appropriate rounding mode for

each operation (one operation which is the first bound in the interval result rounded to $-\infty$ and the other to $+\infty$). The logic circuit of the pre-processing unit is shown in the following figure.

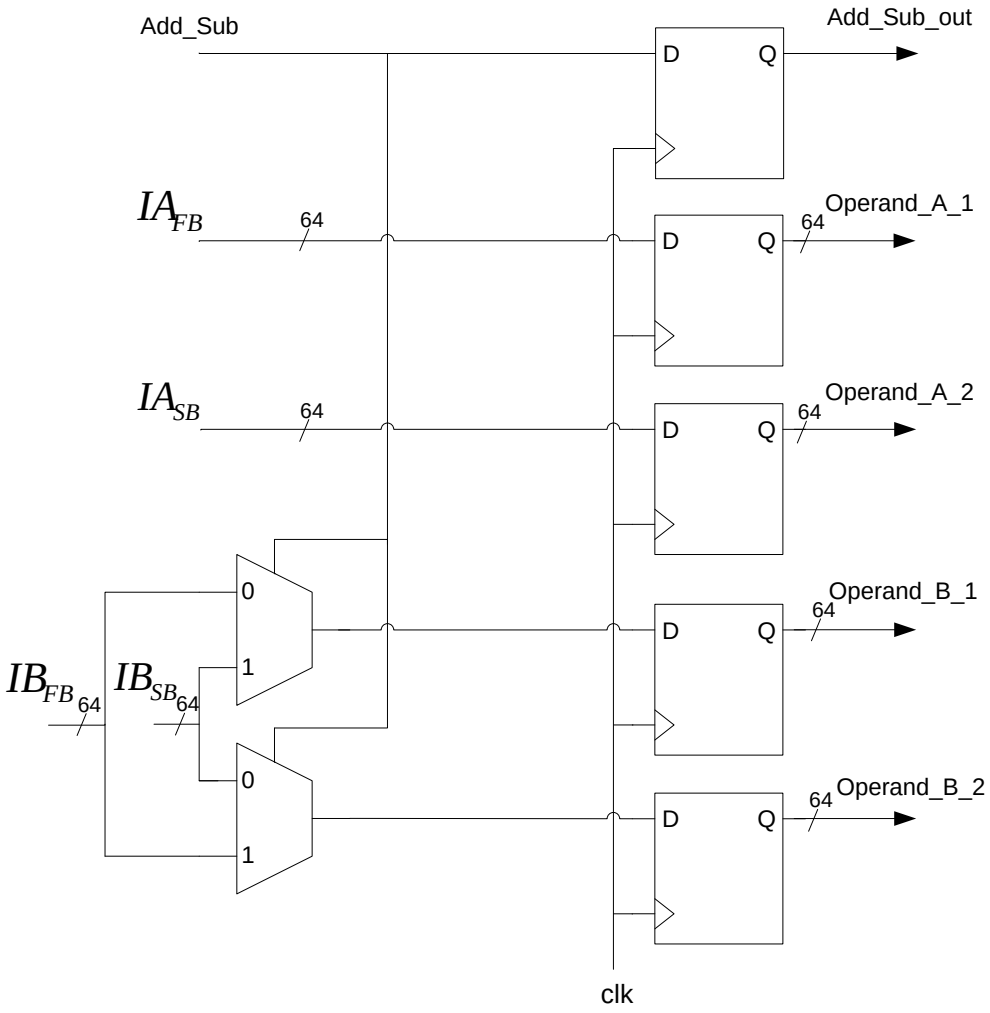

Figure 3.6: pre-processing unit logic circuit

BFP Adder/Subtractors:

They are normal double floating point adder/subtractor units. As mentioned in the Serial Implementation section, the floating point unit was built on a modified VHDL source code posted at opencores site.

3.1.2.2.2 Pipeline stages

The following figure shows a schematic for the pipeline stages of executing three modal interval addition (or subtraction) operations.

Operation No.	Pipeline Stage									
1	Pre-Process C-1	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7		
2		Pre-Process C-1	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7	
3			Pre-Process C-1	Adder/Sub C-1	Adder/Sub C-2	Adder/Sub C-3	Adder/Sub C-4	Adder/Sub C-5	Adder/Sub C-6	Adder/Sub C-7
Clock Cycle	1	2	3	4	5	6	7	8	9	10

Figure 3.7: Pipeline stages for Parallel MIBFP Adder/Subtractor

As shown, the pre-processing unit applies the inputs to the two floating point adder/subtractor units in one clock cycle (Pre-Process C-1). The floating point adder/subtractors take seven clock cycles to execute (from Adder/Sub C-1 to Adder/Sub C-7). The output of each unit is fed into the corresponding result operand register directly.

3.1.2.2.3 Logic Synthesis

As mentioned before, The modules are written in VHDL and synthesized using Quartus-II Altera software tool (for FPGA) and Synopsis Design Compiler (for ASIC). Two types of Altera FPGAs are used to implement the MIBFP Adder/Subtractor [12]:

1- Cyclone II (lower power, cost and speed)
Device EP2C35F672C6 of Cyclone II Family is used. The results are as shown in Table 3.8:

	Area		Timings		
	No. of LEs	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	1813	741	121	7	1
Parallel MIBFP Adder/Subtractor	3784	1716	118	8	1

Table 3.8: Area and Timings (Parallel MIBFP adder/subtractor – Cyclone II)

2- Stratix III (higher power, cost and speed)
Device EP3SL50F780C2 of Stratix III Family is used. The results are as shown in Table 3.9:

	Area	Timings

	No. of ALUTs	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	1178	745	250	7	1
Parallel MIBFP Adder/Subtractor	2358	1719	250	8	1

Table 3.9: Area and Timings (Parallel MIBFP adder/subtractor – Stratix III)

The ASIC results are as shown in Table 3.10:

	Area (mm2)			Timings		
	Combinational Area (mm^2)	Registers Area (mm^2)	Interconnect Area (mm^2)	Clock Frequency (GHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Adder/Subtractor	0.0055	0.0039	0.0032	1.176	7	1
Parallel MIBFP Adder/Subtractor	0.0113	0.0089	0.0068	1.176	8	1

Table 3.10: Area and Timings (Parllel MIBFP adder/subtractor – Nangate 45nm)

As we notice from the previous tables that the modal interval adder/subtraction pipeline throughput is the same as the normal BFP adder/subtractor. Percentage increase in the area is about 115% of the normal BFP adder/subtractor while percentage increase over the serial modal interval adder/subtractor is about 84%.

Clearly from the above we use two parallel floating point adder/subtractor units to implement modal floating point adder/subtractor thus the modal interval addition/subtraction operation executes as fast as the normal floating point addition/subtraction operation i.e we have interval result each clock cycle (in case of pipelining). This comes on the cost of increasing the area (almost the double) as shown in tables 3.8, 3.9, and 3.10.

Combined results for the serial and parallel designs are in the following table:

	Cyclone II		Stratix III		Nangate 45nm Cell Library	
	Area Increase %	Clock Frequency (MHz)	Area Increase %	Clock Frequency (MHz)	Area Increase %	Clock Frequency (MHz)
BFP Add/Sub	-	121	-	250	-	1176
Serial MIBFP Add/Sub	15.8%	120.5	15.8%	250	16.7%	1176
Parallel MIBFP Add/Sub	115.3%	118	112%	250	114.3%	1176

Table 3.11: Interval Adder/Subtrtactor (Combined Results)

We should notice that, the percentage increases are close for the two devices which are from two different families. The clock frequencies differ from the BFP adder/subtractor for the same device although they should be the same but this is due to variations in the ALTERA CAD tool design rules.

3.2 Modal Interval Double Floating Point Multiplier Implementation

The definition of modal interval multiplication with outward rounding is as follows:

$$A * B = \begin{cases} if\ a_1 \geq 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [\nabla a_1 b_1, \Delta a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 \geq 0, b_2 < 0\ then\ [\nabla a_1 b_1, \Delta a_1 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0\ then\ [\nabla a_2 b_1, \Delta a_2 b_2] \\ if\ a_1 \geq 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [\nabla a_2 b_1, \Delta a_1 b_2] \\ if\ a_1 \geq 0, a_2 < 0, b_1 \geq 0, b_2 \geq 0\ then\ [\nabla a_1 b_1, \Delta a_2 b_1] \\ if\ a_1 \geq 0, a_2 < 0, b_1 \geq 0, b_2 < 0 \\ \quad then\ [max(\nabla a_1 b_1, \nabla a_2 b_2), min(\Delta a_2 b_1, \Delta a_1 b_2)] \\ if\ a_1 \geq 0, a_2 < 0, b_1 < 0, b_2 \geq 0\ then\ [0,0] \\ if\ a_1 \geq 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [\nabla a_2 b_2, \Delta a_1 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 \geq 0, b_2 \geq 0\ then\ [\nabla a_1 b_2, \Delta a_2 b_2] \\ if\ a_1 < 0, a_2 \geq 0, b_1 \geq 0, b_2 < 0\ then\ [0,0] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 \geq 0 \\ \quad then\ [min(\nabla a_2 b_1, \nabla a_1 b_2), max(\Delta a_1 b_1, \Delta a_2 b_2)] \\ if\ a_1 < 0, a_2 \geq 0, b_1 < 0, b_2 < 0\ then\ [\nabla a_2 b_1, \Delta a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 \geq 0, b_2 \geq 0\ then\ [\nabla a_1 b_2, \Delta a_2 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 \geq 0, b_2 < 0\ then\ [\nabla a_2 b_2, \Delta a_2 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 \geq 0\ then\ [\nabla a_1 b_2, \Delta a_1 b_1] \\ if\ a_1 < 0, a_2 < 0, b_1 < 0, b_2 < 0\ then\ [\nabla a_2 b_2, \Delta a_1 b_1] \end{cases}$$

Figure 3.8: Outward rounded Modal Interval Multiplication

The differences, from hardware point of view, between classical and modal interval multiplications are

1- It should be noted that cases 1, 3, 4, 9, 10, 11, 12, 14 and 15 are exactly the same as in classical multiplication. The other 7 cases one or the two operands are pure modal intervals. This ensure that modal interval arithmetic is the generalization form of classical interval arithmetic

2- According to Theorems 4.5, 4.8 and 4.9 in [5], the DUAL operator may be used in multiplication operation

3- As mentioned before, outer rounding cannot assure the enclosure of all results so inner rounding may be used

As we mentioned before, we need outer and inner rounding. As we can express inner rounding in terms of outer rounding thus inner rounding may be resolved by the compiler like the outer rounding.

From the above discussion, the only difference between modal interval double precision floating point multiplier and that of the classical is the new 7 cases added in modal interval multiplication.

3.2.1 Handling Infinities in input intervals

One or the two bounds of the input operands may contain $\pm\infty$.This is called extended modal interval multiplication. Table 3.12 writes down all the cases of inputs including $\pm\infty$ (except the cases that one of the interval bounds or the two bounds are zeros) for modal interval multiplication. Note that the shaded part coincides with the classical cases as in [8]. The cases that one of the interval bounds or the two bounds are zeros follow the same rules specified in multiplication cases. The only difference (in these cases) from the normal floating point multiplication that one of the operands is $\pm\infty$ and the other is zero. In that case we will have 0 as the result instead of Nan as stated in the IEEE-754 floating point standard [10].

$$0 * \infty = 0$$

This is the only rule that goes beyond the IEEE-754 standard as mentioned in [8].

From modal interval multiplication equations, we can say that the modal interval multiplication consists of two or four normal floating point multiplication operations rounded to $-\infty$ and $+\infty$ respectively. Thus handling infinities in the modal interval multiplication coincide with handling infinities in the normal floating point multiplication with the proper rounding except for

cases 7, 10 in modal interval multiplication which result in degenerate interval with zero value. These two cases need to generate intervals with zero bounds so they need special handling in the implementation.

Table 3.13 shows the floating point multiplication including $\pm\infty$ in inputs according to the IEEE-754 standard [10].

Multiplication	$[b_1, b_2]$ $b_1<0, b_2<0$	$[b_1, b_2]$ $b_1<0, b_2>0$	$[b_1, b_2]$ $b_1>0, b_2>0$	$(-\infty, b_2]$, $b_2<0$	$(-\infty, b_2]$, $b_2>0$	$[b_1, +\infty)$, $b_1<0$	$[b_1, +\infty)$, $b_1>0$	$(-\infty,+\infty)$	$[b_1, b_2]$ $b_1>0, b_2<0$	$[b_1, -\infty)$, $b_1<0$	$[b_1, -\infty)$, $b_1>0$	$(+\infty, b_2]$, $b_2<0$	$(+\infty, b_2]$, $b_2>0$	$(+\infty, -\infty)$
$[a_1, a_2]$ $a_1<0,\ a_2<0$	$[a_2b_2, a_1b_1]$	$[a_1b_2, a_1b_1]$	$[a_1b_2, a_2b_1]$	$[a_2b_2, +\infty)$	$[a_1b_2, +\infty)$	$(-\infty, a_1b_1]$	$(-\infty, a_2b_1]$	$(-\infty, +\infty)$	$[\ a_2b_2, a_2b_1]$	$(-\infty,a_2b_1]$	$(+\infty,a_2b_1]$	$[a_2b_2, -\infty)$	$[a_1b_2, -\infty)$	$(+\infty, -\infty)$
$[a_1, a_2]$ $a_1<0,\ a_2>0$	$[a_2b_1, a_1b_1]$	$[\min(a_1b_2, a_2b_1),$ $\max(a_1b_1, a_2b_2)]$	$[a_1b_2, a_2b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$[a_2b_1,a_1b_1]$	$[0, 0]$	$[0, 0]$	$[a_1b_2,a_2b_2]$	$[0, 0]$
$[a_1, a_2]$ $a_1>0,\ a_2>0$	$[a_2b_1, a_1b_2]$	$[a_2b_1, a_2b_2]$	$[a_1b_1, a_2b_2]$	$(-\infty, a_1b_2]$	$(-\infty, a_2b_2]$	$[a_2b_1,+\infty)$	$[a_1b_1,+\infty)$	$(-\infty, +\infty)$	$[a_1b_1, a_1b_2]$	$[a_2b_1, -\infty)$	$[a_1b_1,-\infty)$	$(+\infty,a_1b_2]$	$(+\infty,a_2b_2]$	$(+\infty, -\infty)$
$(-\infty, a_2]$, $a_2<0$	$[a_2b_2, +\infty)$	$(-\infty, a_2b_1]$	$[a_2b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2b_1]$	$(-\infty, a_2b_2]$	$(-\infty, +\infty)$	$[\ a_2b_2, a_2b_1]$	$(+\infty,+\infty)$	$(+\infty,a_2b_1]$	$[a_2b_2, -\infty)$	$(-\infty, -\infty)$	$(+\infty, -\infty)$
$(-\infty, a_2]$, $a_2>0$	$[a_2b_1, +\infty)$	$(-\infty, a_2b_2]$	$[a_1b_1, a_2b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$[a_2b_1,+\infty)$	$[0, 0]$	$[0, 0]$	$(-\infty,a_2b_2]$	$[0, 0]$
$[a_1, +\infty)$, $a_1<0$	$(-\infty, a_1b_1]$	$[a_1b_2, +\infty)$	$[a_2b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1b_1,+\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$(-\infty,a_1b_1]$	$[0, 0]$	$[0, 0]$	$[a_1b_2, +\infty)$	$[0, 0]$
$[a_1, +\infty)$, $a_1>0$	$(-\infty, a_1b_2]$	$[a_1b_1, +\infty)$	$(-\infty, a_1b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1b_1,+\infty)$	$[a_1b_1,+\infty)$	$(-\infty, +\infty)$	$[a_1b_1, a_1b_2]$	$(-\infty, -\infty)$	$[a_1b_1,-\infty)$	$(+\infty,a_1b_2]$	$(+\infty, +\infty)$	$(+\infty, -\infty)$
$(-\infty,+\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$[0, 0]$	$[0, 0]$	$(-\infty, +\infty)$	$[0, 0]$
$[a_1, a_2]$ $a_1>0,\ a_2<0$	$[a_2b_2, a_1b_1]$	$[0, 0]$	$[a_1b_1, a_2b_1]$	$[a_2b_2, a_1b_2]$	$[0, 0]$	$[0, 0]$	$[a_1b_1,a_2b_1]$	$[0, 0]$	$[\max(a_1b_1, a_2b_2),$ $\min(a_1b_2, a_2b_1)]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$
$[a_1, -\infty)$, $a_1<0$	$(-\infty, a_1b_1]$	$[a_1b_2, a_1b_1]$	$[a_1b_2, -\infty)$	$(+\infty, +\infty)$	$[a_1b_2,+\infty)$	$(-\infty, a_1b_1]$	$(-\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty,a_1b_1]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$[a_1b_2, -\infty)$	$(+\infty, -\infty)$
$[a_1, -\infty)$, $a_1>0$	$(+\infty, a_1b_2]$	$[0, 0]$	$[a_1b_1, -\infty)$	$(+\infty, a_1b_2]$	$[0, 0]$	$[0, 0]$	$[a_1b_1, -\infty)$	$[0, 0]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$
$(+\infty, a_2]$, $a_2<0$	$[a_2b_2, -\infty)$	$[0, 0]$	$(+\infty, a_1b_1]$	$[a_2b_2, -\infty)$	$[0, 0]$	$[0, 0]$	$(+\infty, -\infty)$	$[0, 0]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$
$(+\infty, a_2]$, $a_2>0$	$[a_2b_1, -\infty)$	$[a_2b_1, a_2b_2]$	$(+\infty, a_1b_2]$	$(-\infty, -\infty)$	$[a_2b_2, +\infty)$	$[a_2b_1, +\infty)$	$(+\infty,+\infty)$	$(-\infty, +\infty)$	$(+\infty, -\infty)$	$[a_2b_1, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, a_2b_2]$	$(+\infty, -\infty)$
$(+\infty, -\infty)$	$(+\infty, -\infty)$	$[0, 0]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$[0, 0]$	$[0, 0]$	$(+\infty, -\infty)$	$[0, 0]$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$	$(+\infty, -\infty)$

Table 3.12: Extended Modal Interval Multiplication

Multiplication	$-\infty$	B < 0	0	B > 0	$+\infty$
$-\infty$	$+\infty$	$+\infty$	Nan	$-\infty$	$-\infty$
A < 0	$+\infty$	A×B	0	A×B	$-\infty$
0	Nan	0	0	0	Nan
A > 0	$-\infty$	A×B	0	A×B	$+\infty$
$+\infty$	$-\infty$	$-\infty$	Nan	$+\infty$	$+\infty$

Table 3.13: Floating Point Multiplication

3.2.2 Hardware Implementation

The problem in modal interval multiplication that dependent on the signs of the input interval bounds, the interval result calculations change as stated before. Besides, cases 6 and 11 need four multiplication operations and two comparisons instead of two multiplication operations as in all other cases.

It can be shown that the four multiplication operations in the two special cases can be reduced to three multiplication operations and four comparisons [17],[18]. Many algorithms are proposed for classical and modal interval multiplication in [17]. These algorithms are suitable for software. We present here two hardware implementations. Serial interval multiplier which uses one double floating point multiplier and the other one is parallel interval multiplier which uses two double floating point multipliers. Serial interval multiplier minimizes the area as it uses one floating point multiplier while parallel interval multiplier maximizes the area as it uses two floating point multipliers. The details of each implementation are in the next sections.

3.2.2.1 Parallel Interval Multiplier

As we mentioned before based on sign distinction of the two interval bounds of each input interval, the multiplication operation can be reduced to two multiplications except for cases 6 and 11 of the multiplication cases.

Let the two input intervals $A = [a_1, a_2], B = [b_1, b_2]$ and
the interval result $R = A \times B = [r_1, r_2]$ then

$$If\ a_1 \geq 0\ then\ x_3 = 0\ else\ x_3 = 1$$
$$If\ a_2 \geq 0\ then\ x_2 = 0\ else\ x_2 = 1$$
$$If\ b_1 \geq 0\ then\ x_1 = 0\ else\ x_1 = 1$$
$$If\ b_2 \geq 0\ then\ x_0 = 0\ else\ x_0 = 1$$

$x_3, x_2, x_1 and\ x_0$ are simply flags that represent the signs of the intervals bounds $a_1, a_2, b_1\ and\ b_2$ respectively. These flags are used to express the interval multiplication in terms of the signs of the four input floating point numbers ($a_1, a_2, b_1\ and\ b_2$) as shown in the following table.

#	$x_3x_2x_1x_0$	r_1	r_2
1	0000	$\nabla a_1 b_1$	$\Delta a_2 b_2$
2	0001	$\nabla a_1 b_1$	$\Delta a_1 b_2$
3	0010	$\nabla a_2 b_1$	$\Delta a_2 b_2$
4	0011	$\nabla a_2 b_1$	$\Delta a_1 b_2$
5	0100	$\nabla a_1 b_1$	$\Delta a_2 b_1$
6	0101	$max(\nabla a_1 b_1, \nabla a_2 b_2)$	$min(\Delta a_1 b_2, \Delta a_2 b_1)$
7	0110	0	0
8	0111	$\nabla a_2 b_2$	$\Delta a_1 b_2$
9	1000	$\nabla a_1 b_2$	$\Delta a_2 b_2$
10	1001	0	0
11	1010	$min(\nabla a_1 b_2, \nabla a_2 b_1)$	$max(\Delta a_1 b_1, \Delta a_2 b_2)$
12	1011	$\nabla a_2 b_1$	$\Delta a_1 b_1$
13	1100	$\nabla a_1 b_2$	$\Delta a_2 b_1$
14	1101	$\nabla a_2 b_2$	$\Delta a_2 b_1$
15	1110	$\nabla a_1 b_2$	$\Delta a_1 b_1$
16	1111	$\nabla a_2 b_2$	$\Delta a_1 b_1$

Table 3.14: Interval Multiplication in terms of bounds' signs

In case of parallel design, we have two floating point multipliers. Assuming that y_1, y_2, y_3, y_4 are the input operands of the two multipliers successively and z_1, z_2 are the outputs of the multipliers respectively then we can rewrite the above table as following.

#	$x_3x_2x_1x_0$	y_1	y_2	z_1	y_3	y_4	z_2	r_1	r_2
1	0000	a_1	b_1	∇	a_2	b_2	Δ	z_1	z_2
2	0001	a_1	b_1	∇	a_1	b_2	Δ	z_1	z_2
3	0010	a_2	b_1	∇	a_2	b_2	Δ	z_1	z_2
4	0011	a_2	b_1	∇	a_1	b_2	Δ	z_1	z_2
5	0100	a_1	b_1	∇	a_2	b_1	Δ	z_1	z_2
6	0101	a_1	b_2 b_1	Δ ∇	a_2	b_1 b_2	Δ ∇	- $max(z_1, z_2)$	$min(z_1, z_2)$ -
7	0110	0	0	0	0	0	0	z_1	z_2
8	0111	a_2	b_2	∇	a_1	b_2	Δ	z_1	z_2
9	1000	a_1	b_2	∇	a_2	b_2	Δ	z_1	z_2
10	1001	0	0	0	0	0	0	z_1	z_2
11	1010	a_1	b_2 b_1	∇ Δ	a_2	b_1 b_2	∇ Δ	$min(z_1, z_2)$ -	- $max(z_1, z_2)$
12	1011	a_2	b_1	∇	a_1	b_1	Δ	z_1	z_2
13	1100	a_1	b_2	∇	a_2	b_1	Δ	z_1	z_2
14	1101	a_2	b_2	∇	a_2	b_1	Δ	z_1	z_2
15	1110	a_1	b_2	∇	a_1	b_1	Δ	z_1	z_2
16	1111	a_2	b_2	∇	a_1	b_1	Δ	z_1	z_2

Table 3.15: Inputs and outputs for each floating point multiplier
in terms of bounds' signs
(**∇ or Δ represents only rounding mode for the operation**)

Table 3.15 is the same as 3.14 except for cases 6 & 11. In these two cases we have four multiplications so we can do two multiplication operations in one clock cycle and the other two multiplications in the second clock cycle. To maximize the efficiency we had to reformulate the input operands such that we have the first result bound calculated first then the second result bound calculated in the next cycle. This reformulation has the advantage that the two comparisons are done in two different cycles thus we can use only one comparator instead of two without increasing the number of clock cycles.

By using the simple and effective Karnaugh map method, we can implement the circuitry to get y_1, y_2, y_3, y_4 keeping in mind that a special circuitry need to be added for cases 6 and 11 to handle the other two multiplication operations and the comparison before obtaining the final result.

There are some issues we should take into account in interval

multiplication. One issue we should note that the sign distinction is not as simple as checking the sign bit of each number. The IEEE-754 floating point standard [10] states that there are two zeros (+0, -0) with the double formats different in sign bit as following:

0+	0	0…….0	000…….0
0-	1	0…….0	000…….0

Accordingly when we say $a_1 \geq 0$, we mean that $a_1 \geq +0$ & $a_1 \geq -0$ so we must take into account (-0) when we implement sign distinction. Another issue related to IEEE-754 floating point standard that when we have $\pm\infty$ in one of the bounds of the input intervals, the rules will be as that of the normal floating point multiplications except for two cases. The first case when we have $(\infty \times 0)$ which gives Nan in the IEEE-754 standard [10] but in interval multiplication it gives 0 as mentioned in [8] (for the classical case) and as implemented in the software Libraries (Intlab for classical intervals and ivalDb for modal intervals) [14],[15]. For that reason the double floating point multiplier must be modified to override this case when we do interval multiplication. A flag is added to the two floating point multipliers to distinguish between the normal case and the interval one. The last issue is about the cases that result in degenerate intervals with zero value (cases 7 & 10 in Table 3.15). As in Table 3.15, we can simply obtain the zero interval by detecting these cases and apply zeros as input operands for the floating point multipliers. Once again the problem appears when we have $\pm\infty$ in the input intervals. Consider for example

$$(-\infty, 2] \times (+\infty, -5]$$

There is no reference for this issue except the IVALDB library which simply gives the result (Nan, Nan) [15].

3.2.2.1.1 Hardware Architecture

The high level architecture of the Parallel MIBFP Multiplier (Parallel Modal Interval Binary Floating Point Multiplier) is as in Figure 3.9

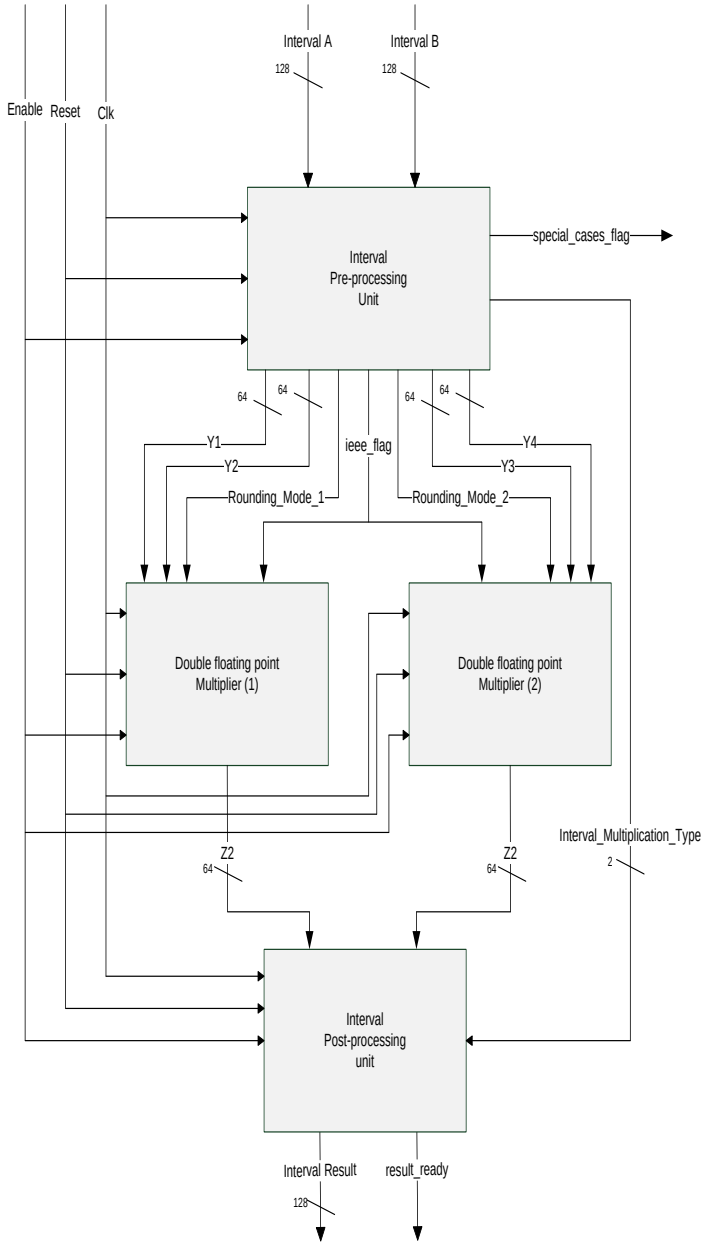

Figure 3.9: Modal Interval Double Floating Point Multiplier (Parallel Implementation)

Interval Pre-processing unit:

Its function is as follows:

1- It divides the interval operands into two parallel floating point multiplication operations with the appropriate rounding mode for each operation.Y1, Y2, Rounding_Mode_1 are inputs to the first

multiplier and Y3,Y4, Rounding_Mode_2 are inputs to the second multiplier.

2- For the two special cases 6, 11 in Table 3.15, the four multiplication operations are divided into two multiplication operations on two cycles. The special_cases_flag output port rises when one of these two cases happens in the input intervals

3- Set ieee_flag to zero to override the case $(\infty \times 0)$ as mentioned in the previous section.

4- It informs the Post-processing unit what the type of the interval multiplication that it will handle. The interval multiplication is divided into three types. One is the normal interval multiplication which consists of two floating point multiplications and the other two are the two special cases (cases 6 and 11 in Table 3.15) that have four multiplication operations.

Logic equations and circuits:

Consider $IA_{FB}, IA_{SB}, IB_{FB}, IB_{SB}$ are first and second bounds of intervals A and B. The following internal signals are constructed using simple AND, OR and NOT logic gates.

$$Mul_{Dist}(3) = \big(IA_{FB}(0) + IA_{FB}(1) + \cdots + IA_{FB}(62)\big) \cdot IA_{FB}(63)$$
$$Mul_{Dist}(2) = \big(IA_{SB}(0) + IA_{SB}(1) + \cdots + IA_{SB}(62)\big) \cdot IA_{SB}(63)$$
$$Mul_{Dist}(1) = \big(IB_{FB}(0) + IB_{FB}(1) + \cdots + IB_{FB}(62)\big) \cdot IB_{FB}(63)$$
$$Mul_{Dist}(0) = \big(IB_{SB}(0) + IB_{SB}(1) + \cdots + IB_{SB}(62)\big) \cdot IB_{SB}(63)$$

$Mul_{Dist}(3), Mul_{Dist}(2), Mul_{Dist}(1)\ and\ Mul_{Dist}(0)$ are representing $x_3, x_2, x_1\ and\ x_0$ (in Table 3.15) respectively.

$$Inf_{IA_{FB}} = \big(IA_{FB}(62) + \cdots + IA_{FB}(52)\big) \cdot \overline{IA_{FB}(51) + \cdots + IA_{FB}(0)}$$
$$Inf_{IA_{SB}} = \big(IA_{SB}(62) + \cdots + IA_{SB}(52)\big) \cdot \overline{IA_{SB}(51) + \cdots + IA_{SB}(0)}$$
$$Inf_{IB_{FB}} = \big(IB_{FB}(62) + \cdots + IB_{FB}(52)\big) \cdot \overline{IB_{FB}(51) + \cdots + IB_{FB}(0)}$$

$$Inf_{IB_{SB}} = \left(IB_{SB}(62) + \cdots + IB_{SB}(52)\right) \cdot \overline{IB_{SB}(51) + \cdots + IB_{SB}(0)}$$

$$nf_{flag} = Inf_{IA_{FB}} + Inf_{IA_{SB}} + Inf_{IB_{FB}} + Inf_{IB_{SB}}$$

$$\begin{aligned} NanResult_{flag} &= Inf_{flag} \\ &+ \left(\overline{Mul_{Dist}(3)} \cdot Mul_{Dist}(2) \cdot Mul_{Dist}(1) \cdot \overline{Mul_{Dist}(0)}\right) \\ &+ \left(Mul_{Dist}(3) \cdot \overline{Mul_{Dist}(2)} \cdot \overline{Mul_{Dist}(1)} \cdot Mul_{Dist}(0)\right) \end{aligned}$$

The Inf_{flag} indicates if one or more of the input interval bounds is $\pm\infty$. The $NanResult_{flag}$ indicates if the result will be Nan or not.

$$SC_{Classical} = Mul_{Dist}(3) \cdot \overline{Mul_{Dist}(2)} \cdot Mul_{Dist}(1) \cdot \overline{Mul_{Dist}(0)}$$
$$SC_{Modal} = \overline{Mul_{Dist}(3)} \cdot Mul_{Dist}(2) \cdot \overline{Mul_{Dist}(1)} \cdot Mul_{Dist}(0)$$
$$SC_{Enable} = SC_{Classical} \cdot SC_{Modal}$$

The SC_{Enable} signal indicates if the input intervals result in one of the two multiplication special cases (cases 6 and 11 in Table 3.15).

Now we can generate the output signals of the interval pre-processing unit in terms of the above signals. The logic circuits of the outputs are presented in the following figures.

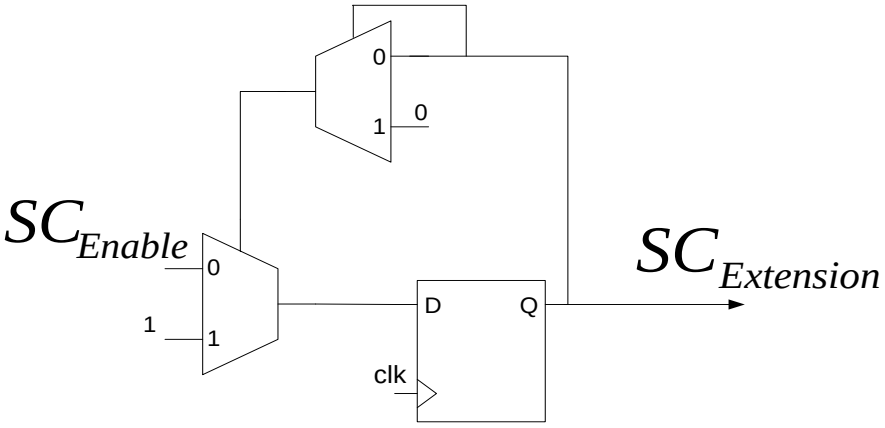

Figure 3.10: Special case extension output signal

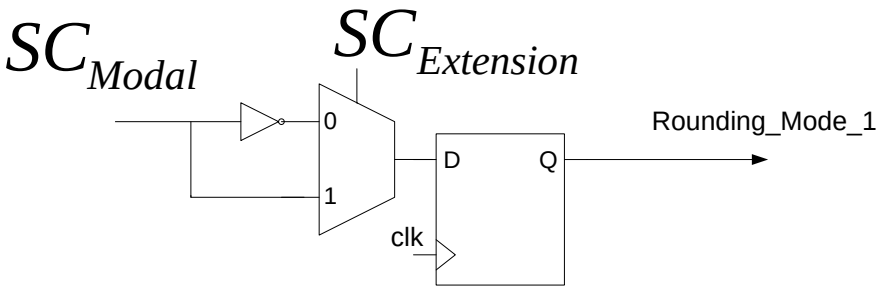

Figure 3.11: Rounding mode (input to the first floating point multiplier)

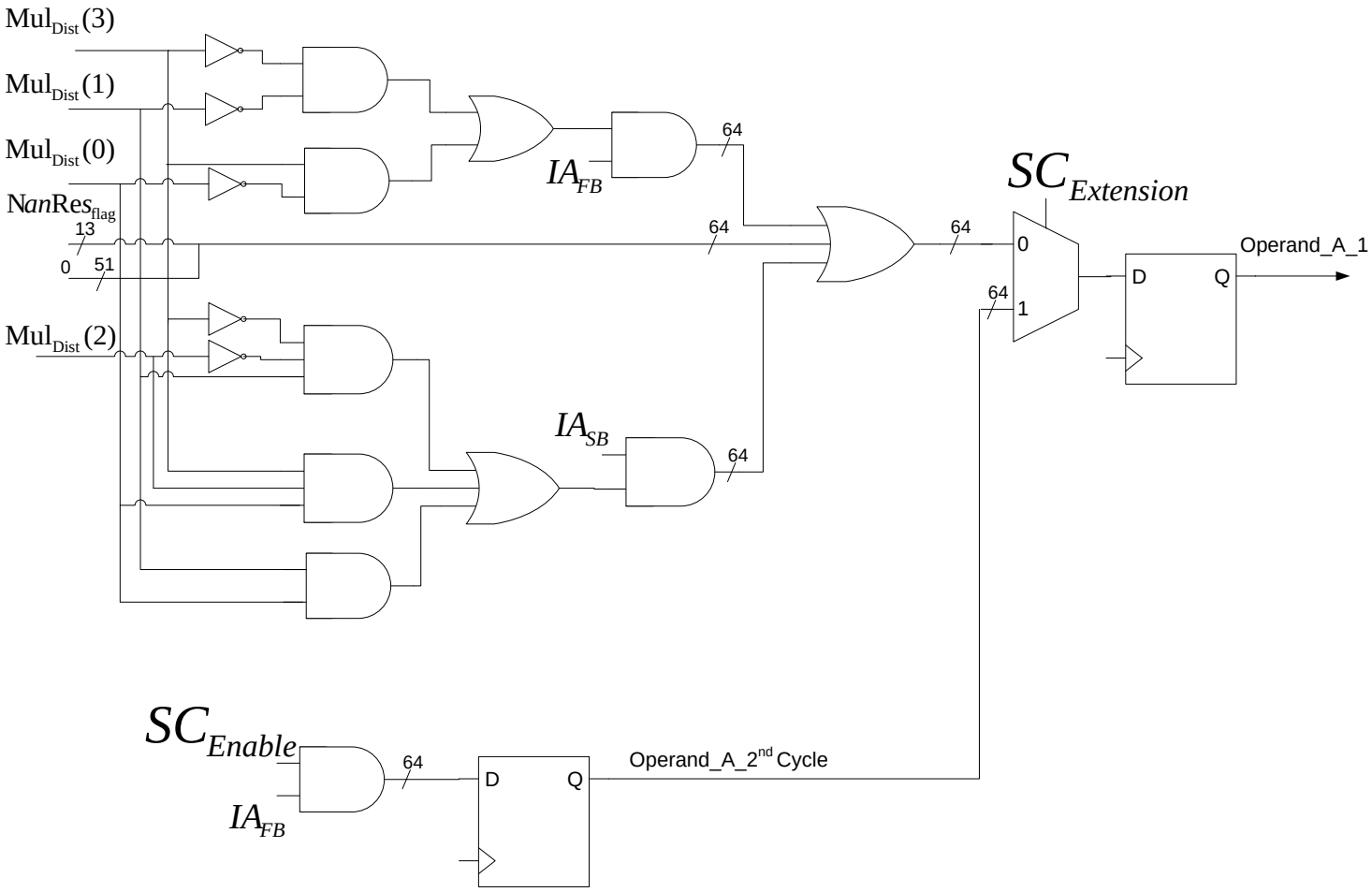

Figure 3.12: Operand_A_1 (operand_A of the first floating point multiplier)

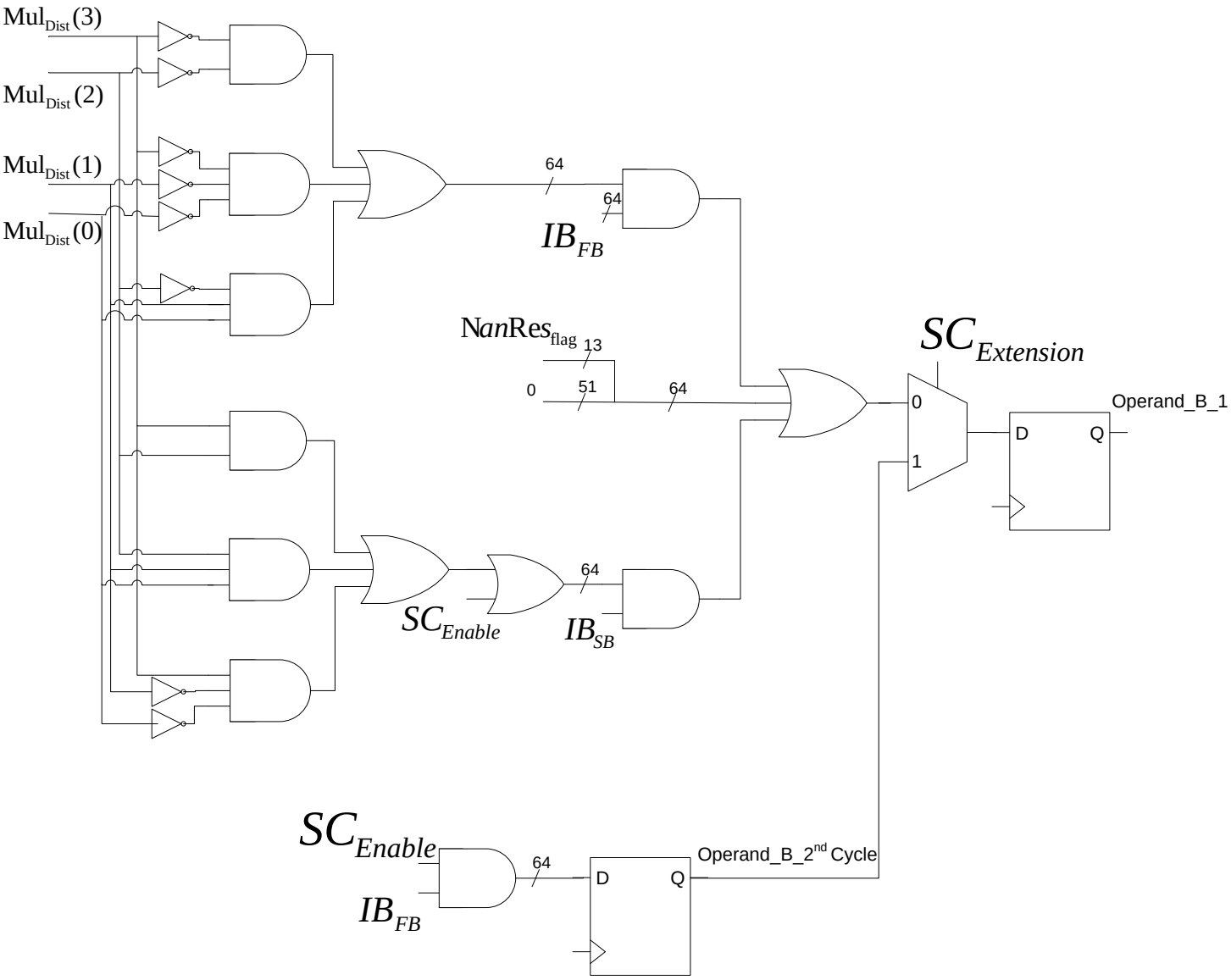

Figure 3.13: Operand_B_1 (operand_B of the first floating point multiplier)

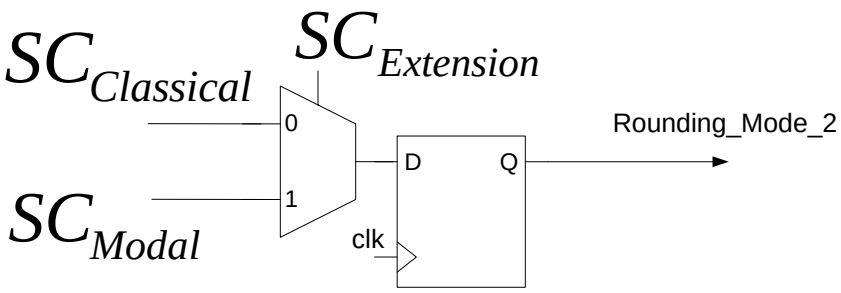

Figure 3.14: Rounding mode (input to the second floating point multiplier)

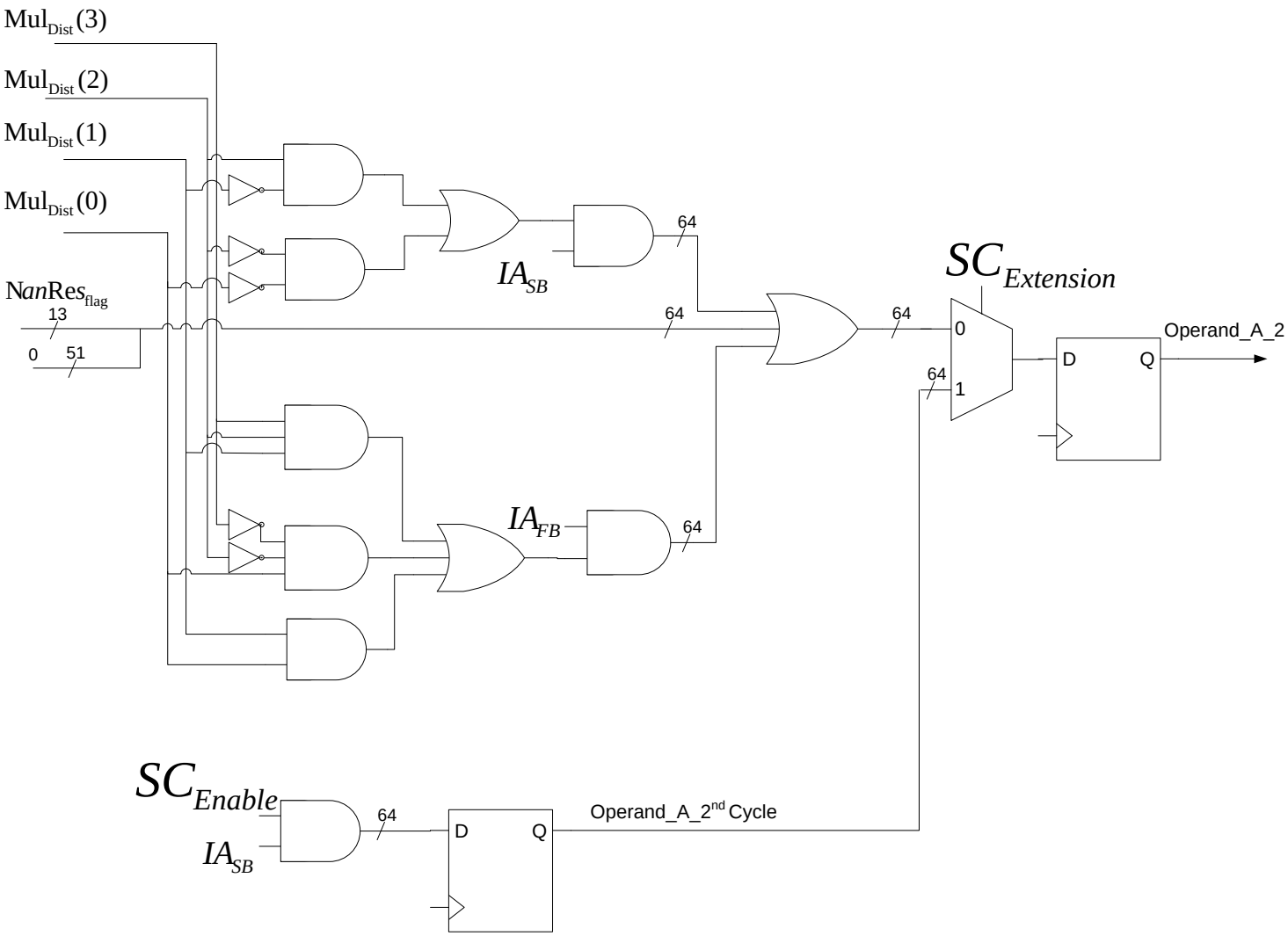

Figure 3.15: Operand_A_2 (operand_A of the second floating point multiplier)

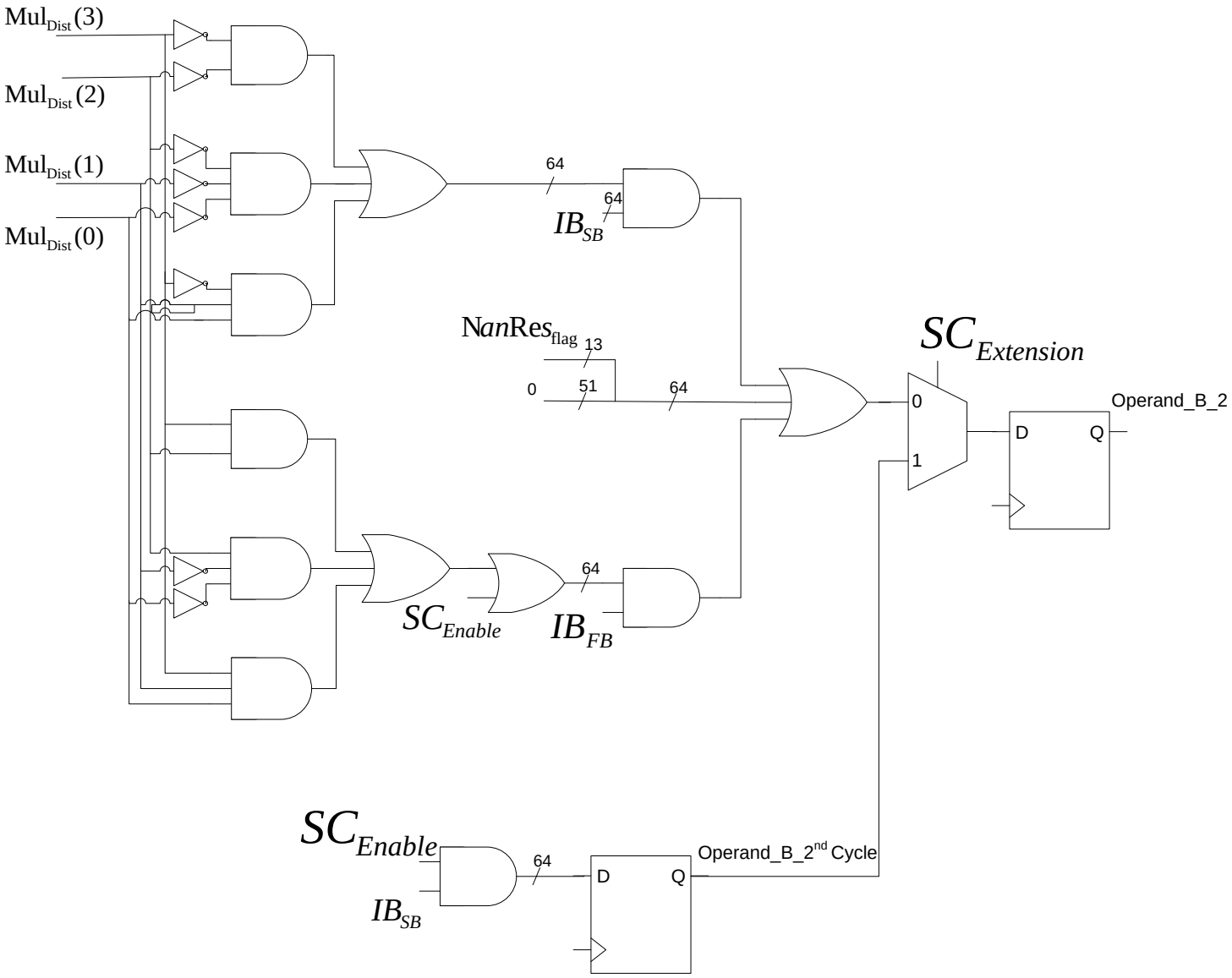

Figure 3.16: Operand_B_2 (operand_B of the second floating point multiplier)

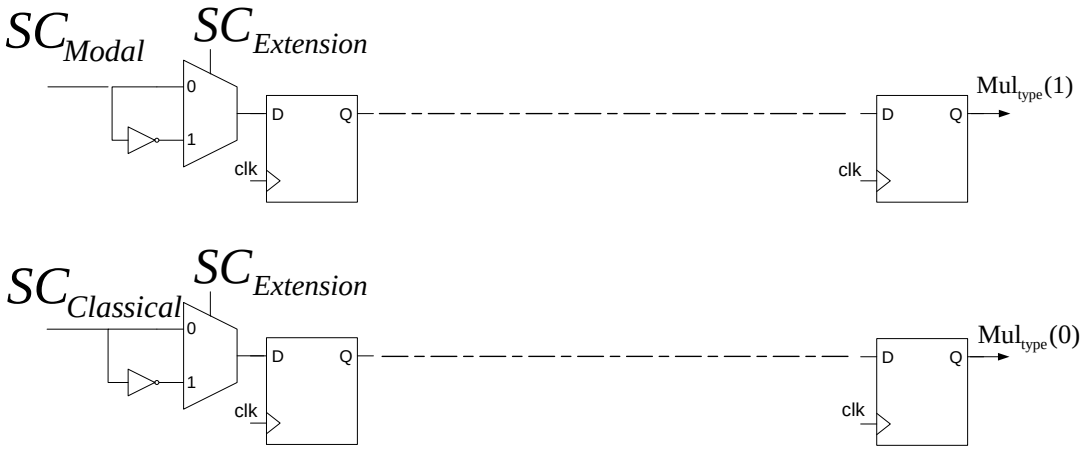

Figure 3.17: Multiplication Type (normal, classical special case or modal special case)

BFP Multipliers:

They are normal double floating point Multiplier units. As in the modal interval adder/subtractor designs, the floating point unit was originally built using the open source VHDL code of the IEEE-754 compliant double precision floating point core posted at opencores site [40]. The VHDL source code of the floating point multiplier was massively modified to fix bugs in pipelining, to handle special cases (like denormalized numbers and infinities), and to reduce the number of pipeline stages. Also, the ieee_flag is added to override the case $(\infty \times 0)$ as mentioned before. The design and implementation of the floating point multiplier unit will not be discussed as this is out of the scope of the thesis topic. However, the area and timings results of this unit are used to compare with the results obtained for the Modal interval multiplier designs.

Interval Post-processing unit:

Its function is as follows:

1- Depending on interval multiplication type, it assigns the interval result bounds. In case of normal interval, the two bounds are the output of the two floating point multipliers. In case of the two special cases, we have four multiplications in two cycles so we have two floating point results per cycle. In the first cycle, the two floating point results are fed into a comparator to decide which result will be output to one of the bounds. In the second cycle, the other two floating point results are fed into the same comparator to decide which result will be output to the other bound.

2- Set the result_ready flag when there is a ready interval result

Logic circuits:

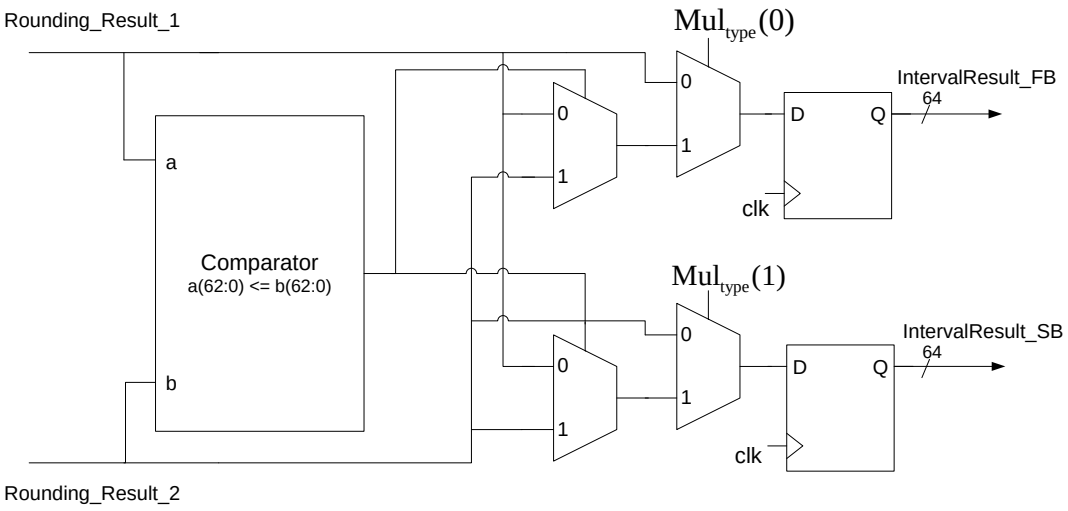

Figure 3.18: Interval result bounds

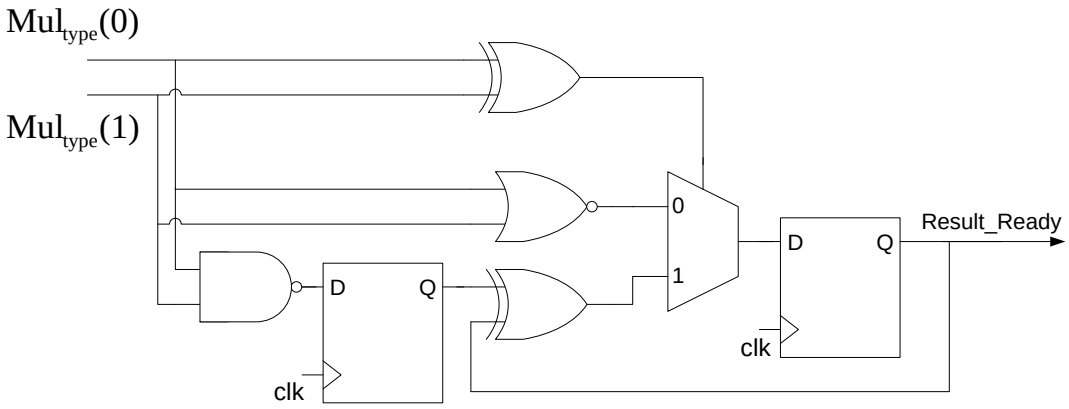

Figure 3.19: Result ready flag logic circuit

As we notice in the logic circuit of interval result bounds (Figure 3.18), we use only one comparator to get the smaller or the bigger value to assign it to one of the interval result bounds (dependent on one of the special cases as in Table 3.15). This comes from that the two inputs of the comparator have equal signs. The only special case for that rule is the case of having (+0) and (-0) as inputs to the comparator. The above figure is a simplified one that doesn't handle this case.

3.2.2.1.2 Pipeline stages

The following figure shows a schematic for the pipeline stages of executing two modal interval multiplication operations. The first operation is a normal case (which takes two multiplication operations) while the second operation is one of the two special cases (which need four multiplication operations).

Operation No.	Pipeline Stage										
1	Pre-Process C-1	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-1		
2		Pre-Process C-1	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-1	
			Pre-Process C-2	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-2
Clock Cycle	1	2	3	4	5	6	7	8	9	10	11

Figure 3.20: Pipeline stages for the Parallel MIBFP Multiplier

As shown, the pre-processing unit applies the inputs to the two floating point multipliers every clock cycle for the normal case and every two clock cycles for the special case. Each floating point multiplier takes seven clock cycles to execute (from MUL C-1 to MUL C-7). The post-processing unit outputs the interval result each clock cycle for the normal case (at Post-Process C-1) or each two clock cycles for the special case (at Post-Process C-2).

3.2.2.1.3 Logic Synthesis

The logic synthesis is done using both FPGA (using ALTERA Quartus-II tool) and ASIC cell-based libraries (using Synopsis Design Compiler tool). For FPGA synthesis; two types of Altera FPGAs are used to implement the MIBFP Multiplier [12]:

1- Cyclone II (lower power, cost and speed)
   Device EP2C35F672C6 of Cyclone II Family is used. The results are as shown in Table 3.16:

	Area			Timings		
	No. of LEs	No. of Embedded Multipliers	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Multiplier	2468	18	1071	128	7	1
Parallel MIBFP Multiplier	6005	36	2793	120	9(10)	1(0.5)

Table 3.16: Area and Timings (Parallel MIBFP multiplier – Cyclone II)

2- Stratix III (higher power, cost and speed)
Device EP3SL50F780C2 of Stratix III Family is used. The results are as shown in Table 3.17:

	Area			Timings		
	No. of ALUTs	No. of Embedded Multipliers	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Multiplier	1484	18	1071	181.5	7	1
Parallel MIBFP Multiplier	3491	36	2793	167	9(10)	1(0.5)

Table 3.17: Area and Timings (Parallel MIBFP multiplier – Stratix III)

The ASIC results are as shown in Table 3.18:

	Area (mm2)			Timings		
	Combinational Area (mm^2)	Registers Area (mm^2)	Interconnect Area (mm^2)	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Multiplier	0.0262	0.0048	0.0106	870	7	1
Parallel MIBFP Multiplier	0.0584	0.0127	0.0237	870	9(10)	1(0.5)

Table 3.18: Area and Timings (Parallel MIBFP multiplier – Nangate 45nm)

As we notice from the previous tables that the modal interval multiplier pipeline throughput is the same as the normal BFP multiplier except for the two special cases (we have half the throughput). Percentage increase in the area is about 148%, 145.5%, and 128% of the normal BFP multiplier for the CYCLONE FPGA, STRATIX FPGA, and ASIC cell library. The first two percentages are quite misleading as we assume that the embedded multipliers and ALUTs have equal area weight which is not correct. Thus percentage increase in case of ASIC cell library is the closest number which is 128%.

Clearly from the above we use two parallel floating point multiplier units to implement modal floating point multiplier thus the modal interval multiplication operation executes as fast as the normal floating point multiplication operation (except for the two special cases) i.e we nearly have one interval result every one clock cycle (in case of pipelining). This comes on the cost of increasing the area almost the double as shown in tables 3.16, 3.17, and 3.18.

3.2.2.2 Serial interval multiplier

Just like in the parallel interval multiplier, by applying case distinction on the bounds of the input intervals the multiplication operation can be reduced to two multiplication operations except for cases 6 & 11. These two cases originally have four multiplication operations but we can reduce them to three multiplications using two comparators as we will see.

Let the two input intervals $A = [a_1, a_2], B = [b_1, b_2]$ and
the interval result $R = A \times B = [r_1, r_2]$ then

$$If\ a_1 \geq 0\ then\ x_3 = 0\ else\ x_3 = 1$$
$$If\ a_2 \geq 0\ then\ x_2 = 0\ else\ x_2 = 1$$
$$If\ b_1 \geq 0\ then\ x_1 = 0\ else\ x_1 = 1$$
$$If\ b_2 \geq 0\ then\ x_0 = 0\ else\ x_0 = 1$$

$$If\ |a_2| \geq |a_1|\ then\ c_0 = 1\ else\ c_0 = 0$$
$$If\ |b_2| \geq |b_1|\ then\ c_1 = 1\ else\ c_1 = 0$$

As we mentioned before, $x_3, x_2, x_1 and\ x_0$ represents the signs of the intervals bounds $a_1, a_2, b_1\ and\ b_2$ respectively. $c_0\ and\ c_1$ are flags that represent the comparison results of the two bounds of each interval. $x_3, x_2, x_1 and\ x_0$ are used to express the interval multiplication in terms of the

signs of the four input floating point numbers ($a_1, a_2, b_1\ and\ b_2$) as shown in Table 3.19. $c_0\ and\ c_1$ are used to reduce the number of multiplications for the cases 6 and 11 from four multiplications to three as shown in Table 3.19.

#	$x_3x_2x_1x_0$		r_1	r_2
1	0000		∇a_1b_1	Δa_2b_2
2	0001		∇a_1b_1	Δa_1b_2
3	0010		∇a_2b_1	Δa_2b_2
4	0011		∇a_2b_1	Δa_1b_2
5	0100		∇a_1b_1	Δa_2b_1
6	0101	c_1c_0		
		00	∇a_1b_1	$min(\Delta a_1b_2, \Delta a_2b_1)$
		01	$max(\nabla a_1b_1, \nabla a_2b_2)$	Δa_1b_2
		10	$max(\nabla a_1b_1, \nabla a_2b_2)$	Δa_2b_1
		11	∇a_2b_2	$min(\Delta a_1b_2, \Delta a_2b_1)$
7	0110		0	0
8	0111		∇a_2b_2	Δa_1b_2
9	1000		∇a_1b_2	Δa_2b_2
10	1001		0	0
11	1010	c_1c_0		
		00	$min(\nabla a_1b_2, \nabla a_2b_1)$	Δa_1b_1
		01	∇a_2b_1	$max(\Delta a_1b_1, \Delta a_2b_2)$
		10	∇a_1b_2	$max(\Delta a_1b_1, \Delta a_2b_2)$
		11	$min(\nabla a_1b_2, \nabla a_2b_1)$	Δa_2b_2
12	1011		∇a_2b_1	Δa_1b_1
13	1100		∇a_1b_2	Δa_2b_1
14	1101		∇a_2b_2	Δa_2b_1
15	1110		∇a_1b_2	Δa_1b_1
16	1111		∇a_2b_2	Δa_1b_1

Table 3.19: Interval Multiplication in terms of bounds' signs & comparisons

The reduction of the four multiplication operations to three can be easily deduced for cases 6 and 11. Consider for example $x_3x_2x_1x_0 = 0101$ which means that $a_1b_1 \geq 0, a_2b_2 \geq 0, a_1b_2 < 0, and\ a_2b_1 < 0$. Now consider that $c_1c_0 = 00$ which means that$|a_2| < |a_1|\ and\ |b_2| < |b_1|$. From this we conclude that $|a_1b_1| < |a_2b_2|$ thus $a_1b_1 < a_2b_2$ so $max(a_1b_1, a_2b_2) = a_1b_1$. The same method applies for all other cases.

In case of the serial design, we have one floating point multiplier. Assuming that y_1, y_2 are the input operands of the multiplier and z_1 is the output of the multiplier then we can rewrite the above table as following:

#	$x_3x_2x_1x_0$		y_1	y_2	z_1	r_1	r_2	t
First Cycle								
1	0000		a_1	b_1	∇	z_1	-	-
2	0001		a_1	b_1	∇	z_1	-	-
3	0010		a_2	b_1	∇	z_1	-	-
4	0011		a_2	b_1	∇	z_1	-	-
5	0100		a_1	b_1	∇	z_1	-	-
6	0101	c_1c_0						
		00	a_1	b_2	Δ	-	-	z_1
		01	a_1	b_1	∇	-	-	z_1
		10	a_1	b_1	∇	-	-	z_1
		11	a_1	b_2	Δ	-	-	z_1
7	0110		0	0	0	z_1	-	-
8	0111		a_2	b_2	∇	z_1	-	-
9	1000		a_1	b_2	∇	z_1	-	-
10	1001		0	0	0	z_1	-	-
11	1010	c_1c_0						
		00	a_1	b_2	∇	-	-	z_1
		01	a_1	b_1	Δ	-	-	z_1
		10	a_1	b_1	Δ	-	-	z_1
		11	a_1	b_2	∇	-	-	z_1
12	1011		a_2	b_1	∇	z_1	-	-
13	1100		a_1	b_2	∇	z_1	-	-
14	1101		a_2	b_2	∇	z_1	-	-
15	1110		a_1	b_2	∇	z_1	-	-
16	1111		a_2	b_2	∇	z_1	-	-
Second Cycle								
1	0000		a_2	b_2	Δ	-	z_1	-
2	0001		a_1	b_2	Δ	-	z_1	-
3	0010		a_2	b_2	Δ	-	z_1	-
4	0011		a_1	b_2	Δ	-	z_1	-
5	0100		a_2	b_1	Δ	-	z_1	-
6	0101	c_1c_0						
		00	a_2	b_1	Δ	-	$min(z_1, t)$	-
		01	a_2	b_2	∇	$max(z_1, t)$	-	-
		10	a_2	b_2	∇	$max(z_1, t)$	-	-

		11	a_2	b_1	Δ	-	$min(z_1, t)$	-
7	0110		0	0	0	-	z_1	-
8	0111		a_1	b_2	Δ	-	z_1	-
9	1000		a_2	b_2	Δ	-	z_1	-
10	1001		0	0	0	-	z_1	-
11	1010	c_1c_0						
		00	a_2	b_1	∇	$min(z_1, t)$	-	-
		01	a_2	b_2	Δ	-	$max(z_1, t)$	-
		10	a_2	b_2	Δ	-	$max(z_1, t)$	-
		11	a_2	b_1	∇	$min(z_1, t)$	-	-
12	1011		a_1	b_1	Δ	-	z_1	-
13	1100		a_2	b_1	Δ	-	z_1	-
14	1101		a_2	b_1	Δ	-	z_1	-
15	1110		a_1	b_1	Δ	-	z_1	-
16	1111		a_1	b_1	Δ	-	z_1	-
Third Cycle								
6	0101	c_1c_0						
		00	a_1	b_1	∇	z_1	-	-
		01	a_1	b_2	Δ	-	z_1	-
		10	a_2	b_1	Δ	-	z_1	-
		11	a_2	b_2	∇	z_1	-	-
11	1010	c_1c_0						
		00	a_1	b_1	Δ	-	z_1	-
		01	a_2	b_1	∇	z_1	-	-
		10	a_1	b_2	∇	z_1	-	-
		11	a_2	b_2	Δ	-	z_1	-

Table 3.20: Inputs and outputs for each floating point multiplier
in terms of bounds' signs & comparisons
(**∇ or Δ represents only rounding mode for the operation**)

As we have only one multiplier, the two bounds are obtained in two cycles for all cases except case 6 and 11 which need three cycles. Table 3.20 shows the inputs and outputs in each cycle. In the two special cases, we had to reorder the three multiplication operations such that we obtain the two results that are fed into the comparator first. This is to have the interval result in three cycles instead if four.

By using the simple and effective Karnaugh map method, we can implement the circuitry to get y_1, y_2 (in each cycle) keeping in mind that a special circuitry need to be added for cases 6 and 11 to handle the third

multiplication operation and the comparison before obtaining the final result.

The serial implementation has the same issues mentioned in the parallel implementation. Accordingly the design should take care of these issues (as in parallel design).

3.2.2.2.1 Hardware Architecture

The high level architecture of the Serial MIBFP Multiplier (Serial Modal Interval Binary Double Floating Point Multiplier) is as in Figure 3.21

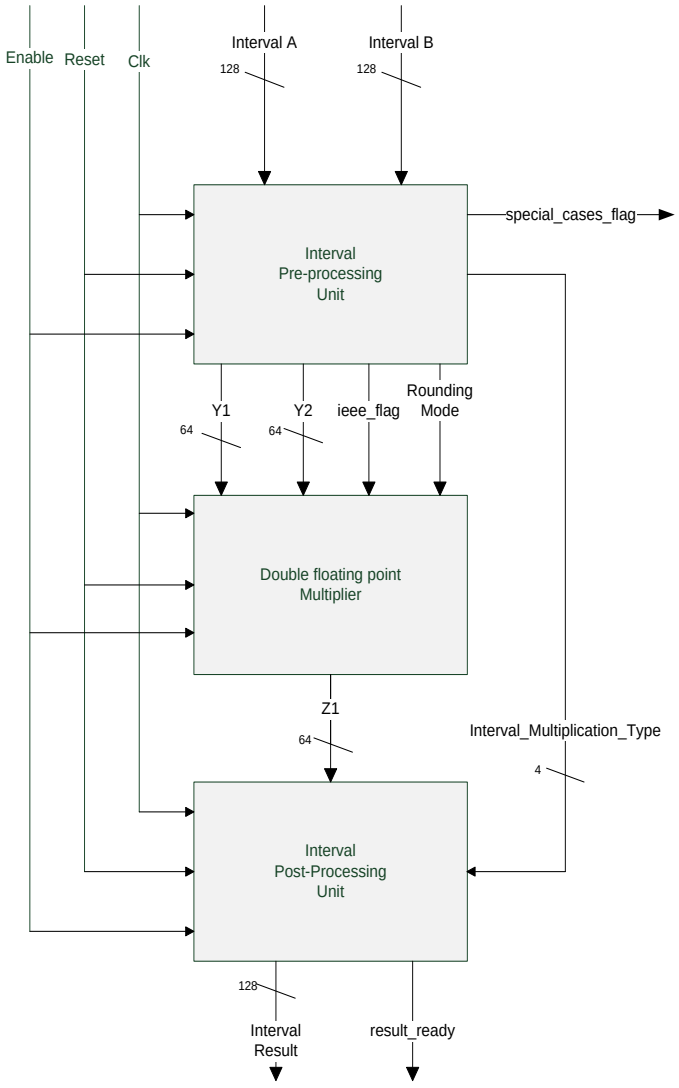

Figure 3.21: Modal Interval Double Floating Point Multiplier (Serial Implementation)

Interval Pre-processing unit:

Its function is as follows:

1- It divides the interval operands into two (or three) sequential floating point multiplication operations with the appropriate rounding mode for each operation.Y1, Y2, Rounding_Mode are inputs to the multiplier.

2- For the two special cases 6 and 11, two comparators are used to reduce the four multiplications to three then set the special_cases_flag output port when one of these two cases happens in the input intervals

3- Set ieee_flag to zero to override the case $(\infty \times 0)$

4- It informs the Post-processing unit about the type of the interval multiplication that it will handle. The interval multiplication is divided into three types. One is the normal interval multiplication which consists of two floating point multiplications and the other two are the two special cases (cases 6 and 11 in Table 3.20) that have three multiplication operations.

Logic equations and circuits:

Consider $IA_{FB}, IA_{SB}, IB_{FB}, IB_{SB}$ are the first and second bounds of intervals A and B. The following internal signals are constructed using simple AND, OR and NOT logic gates.

As mentioned in the parallel interval multiplier section, we need to implement $Mul_{Dist}(3), Mul_{Dist}(2), Mul_{Dist}(1)\ and\ Mul_{Dist}(0)$ flags which are representing $x_3, x_2, x_1\ and\ x_0$ (in Table 3.20) respectively.

$$Cmp\_A = \begin{cases} 1, if\ IA_{FB}(62:0) \leq IA_{SB}(62:0) \\ 0, elsewhere \end{cases}$$

$$Cmp\_B = \begin{cases} 1, if\ IB_{FB}(62:0) \leq IB_{SB}(62:0) \\ 0, elsewhere \end{cases}$$

$Cmp\_A\ and\ Cmp\_B$ are implemented using two 63-bits comparators. They are representing $c_0\ and\ c_1$ signals (in Table 3.20) respectively.

Also we will need to implement the circuitry of the signals Inf_{flag} , $NanResult_{flag}$, $SC_{Classical}$, SC_{Modal} and SC_{Enable} as specified in the Logic circuitry section of the parallel interval multiplier.

The SC_{Enable} signal indicates if the input intervals result in one of the two multiplication special cases (cases 6 and 11 in Table 3.20).

Now we can generate the output signals of the interval pre-processing unit in terms of the above signals. The logic circuits of the outputs are presented in the following figures.

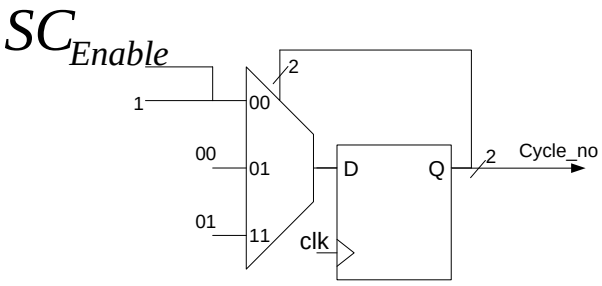

Figure 3.22: Cycle Number

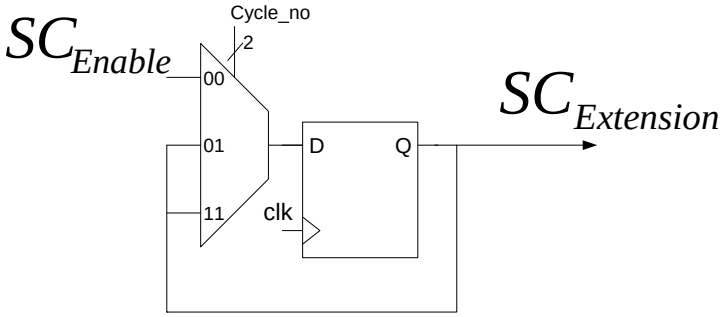

Figure 3.23: Special case extension output signal

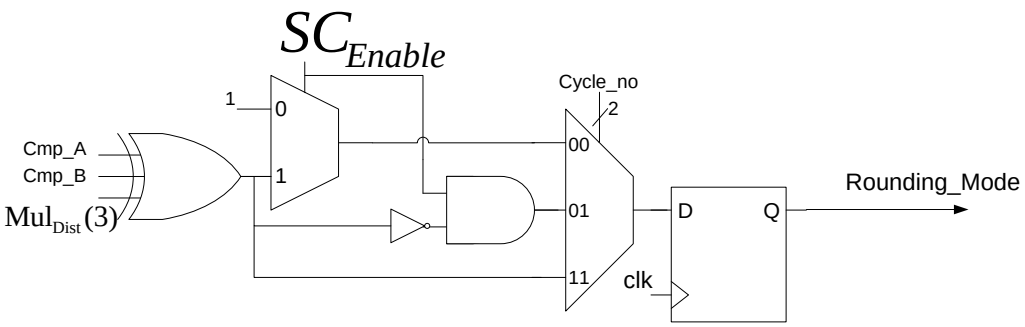

Figure 3.24: Rounding mode logic circuit

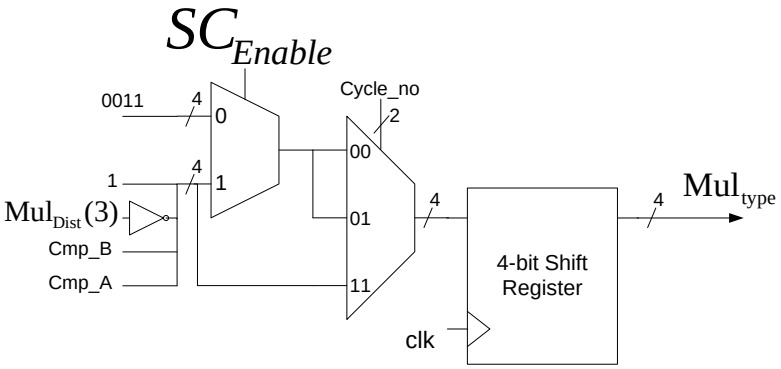

Figure 3.25: Multiplication Type logic circuit

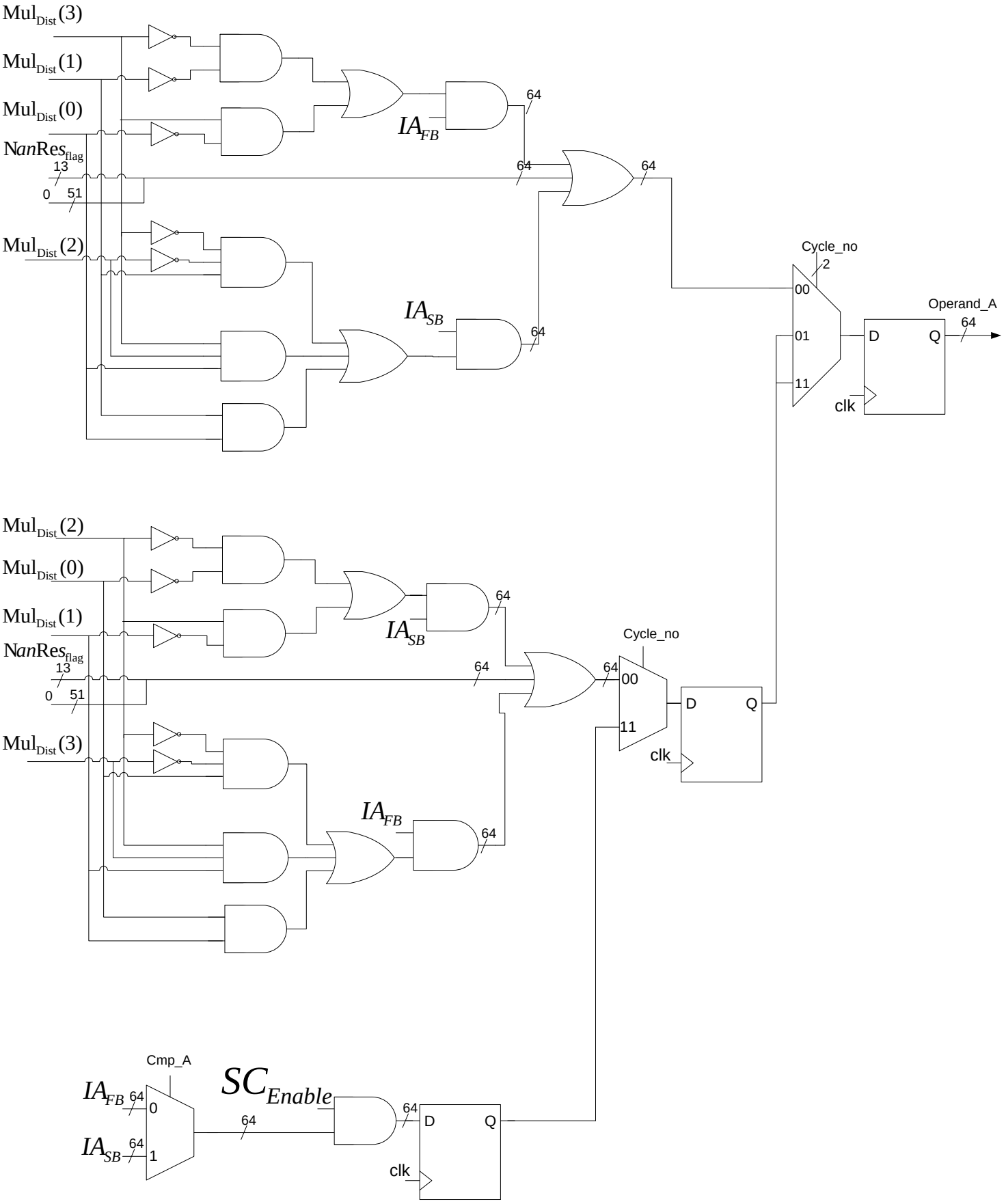

Figure 3.26: Operand_A logic circuit

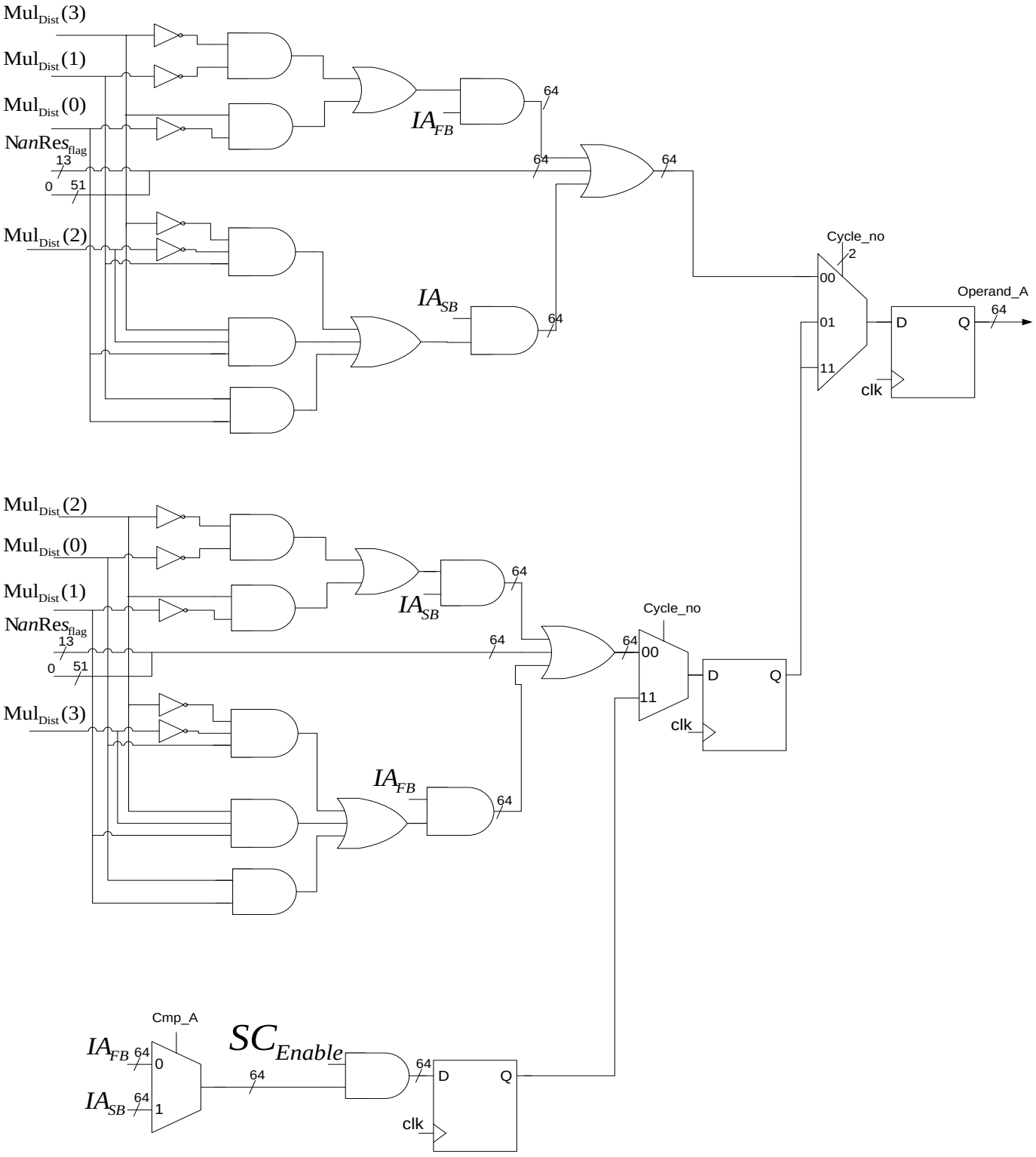

Figure 3.27: Operand_B logic circuit

BFP Multiplier:

It is a normal double floating point Multiplier unit. The ieee_flag is added to override the case $(\infty \times 0)$ as mentioned before.

Interval Post-processing unit:

Its function is as follows:

1- Depending on interval multiplication type, it assigns the interval result bounds. In case of normal interval, the two bounds are the output of the floating point multiplier in two cycles. In case of the two special cases, we have three multiplications in three cycles so we have one floating point results per cycle. In the first cycle, the first floating point result is stored in a temporary register until the second result is ready in the second cycle then both numbers are fed into the comparator to decide which result will be output to one of the bounds. In the third cycle, the last floating point result is fed directly into the other bound.

2- Set the result_ready flag when there is a ready interval result

Logic circuits:

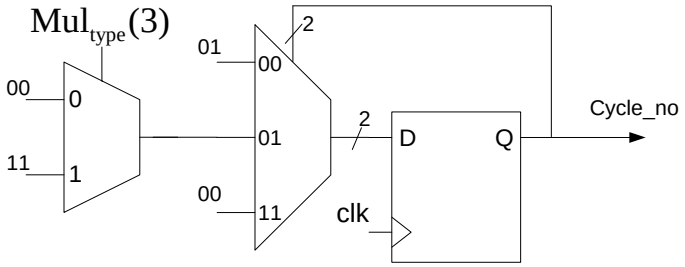

Figure 3.28: Cycle Number

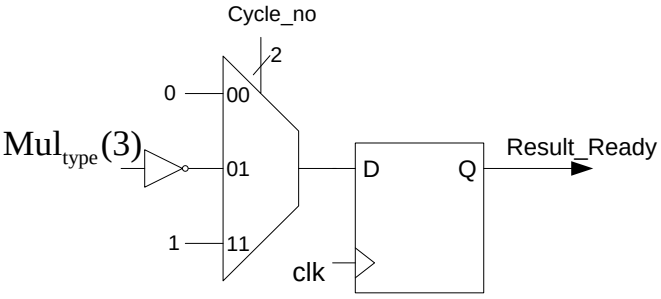

Figure 3.29: Result Ready logic circuit

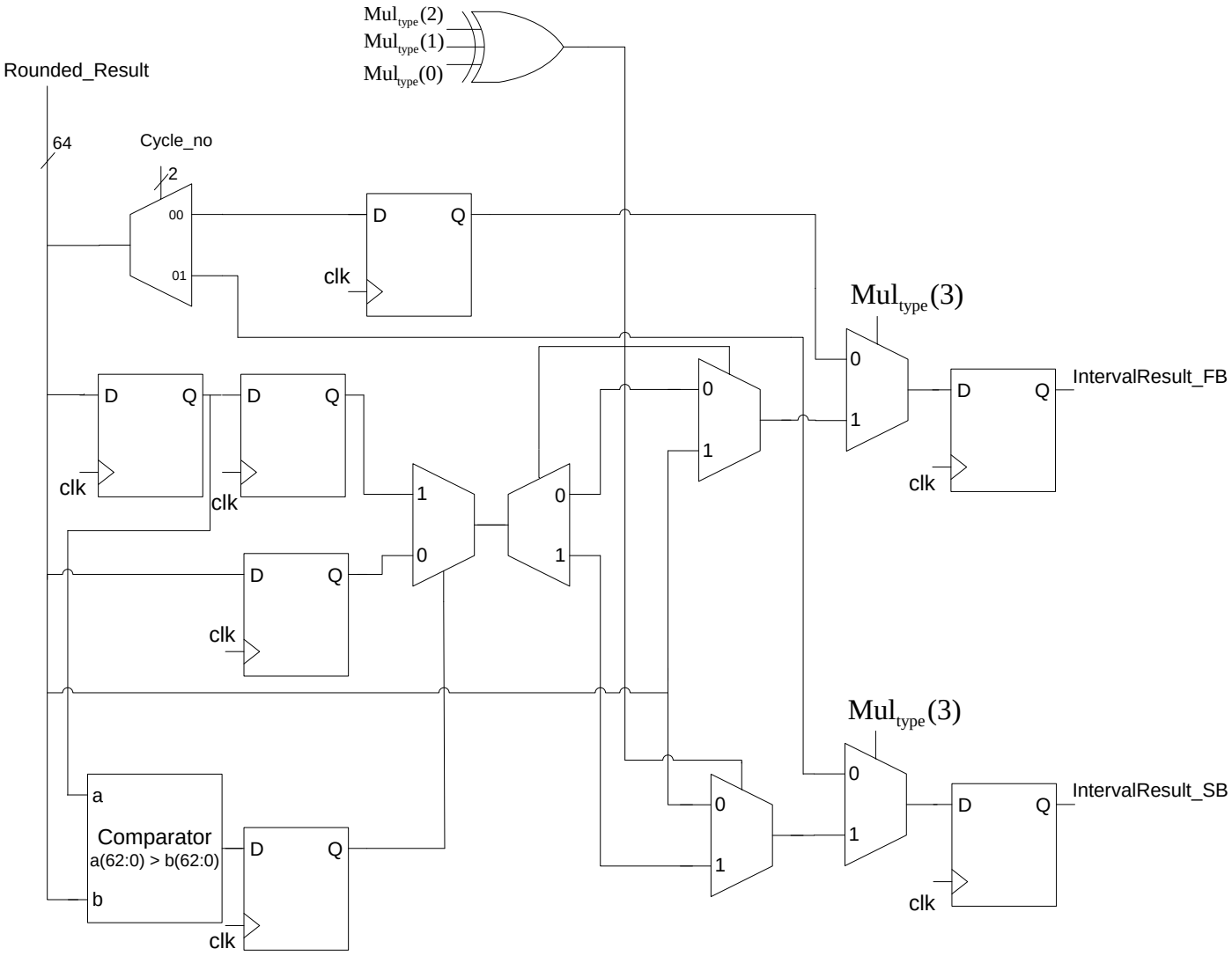

Figure 3.30: Interval result bounds logic circuit

Once again we should notice that we use only one comparator to get the smaller or the bigger value to assign it to one of the interval result bounds (dependent on one of the special cases in Table 3.20). This comes from that the two inputs of the comparator have equal signs. The only special case for that rule is the case of having (+0) and (-0) as inputs to the comparator. The above figure is a simplified one that doesn't handle this case.

3.2.2.2.2 Pipeline stages

The following figure shows a schematic for the pipeline stages of executing two modal interval multiplication operations. The first operation is a normal

case (which takes two multiplication operations) while the second operation is one of the two special cases (which needs four multiplication operations).

Operation No.	Pipeline Stage												
1	Pre-Process C-1	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-1				
		Pre-Process C-2	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-2			
2			Pre-Process C-1	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-1		
				Pre-Process C-2	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-2	
					Pre-Process C-3	MUL C-1	MUL C-2	MUL C-3	MUL C-4	MUL C-5	MUL C-6	MUL C-7	Post-Process C-3
Clock Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 3.31: Pipeline stages for the Serial MIBFP Multiplier

As shown, the pre-processing unit applies the inputs to the floating point multiplier on two clock cycles for the normal case and on three clock cycles for the special case. The floating point multiplier takes seven clock cycles to execute (from MUL C-1 to MUL C-7). The post-processing unit outputs the interval result every two clock cycles for the normal case (at Post-Process C-2) or every three clock cycles for the special case (at Post-Process C-3).

3.2.2.2.3 Logic Synthesis

The logic synthesis is done using both FPGA (using ALTERA Quartus-II tool) and ASIC cell-based libraries (using Synopsis Design Compiler tool). For FPGA synthesis; two types of Altera FPGAs are used to implement the MIBFP Multiplier [12]:

1- Cyclone II (lower power, cost and speed)
Device EP2C35F672C6 of Cyclone II Family is used. The results are as shown in Table 3.21:

	Area			Timings		
	No. of LEs	No. of Embedded Multipliers	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Multiplier	2468	18	1071	128	7	1
Serial MIBFP Multiplier	3648	18	1818	124.5	10(11)	0.5(0.33)

Table 3.21: Area and Timings (Serial MIBFP multiplier – Cyclone II)

2- Stratix III (higher power, cost and speed)
Device EP3SL50F780C2 of Stratix III Family is used. The results are as shown in Table 3.22:

	Area			Timings		
	No. of LEs	No. of Embedded Multipliers	No. of Registers	Clock Frequency (MHz)	Pipeline Depth (Cycles)	Pipeline Throughput
BFP Multiplier	1484	18	1071	181.5	7	1
Serial MIBFP Multiplier	2319	18	1818	177	10(11)	0.5(0.33)

Table 3.22: Area and Timings (Serial MIBFP multiplier – Stratix III)

The ASIC results are as shown in Table 3.23:

	Area (mm2)			Timings		
	Combinational Area (mm^2)	Registers Area (mm^2)	Interconnect Area (mm^2)	Clock Frequency (MHz)	Pipeline Depth	Pipeline Throughput
BFP Multiplier	0.0262	0.0048	0.0106	870	7	1
Serial MIBFP Multiplier	0.0311	0.0082	0.0132	870	10(11)	0.5(0.33)

Table 3.23: Area and Timings (Serial MIBFP multiplier – Nangate 45nm)

As we notice from the previous tables that the pipeline throughput is 1 result per two clock cycles which is half the throughput of normal floating point multiplier except for the two special cases (throughput is one-third that of the normal floating point multiplier).

Percentage increase in the area is about 54%, 61.5%, and 26% of the normal BFP multiplier for the CYCLONE FPGA, STRATIX FPGA, and ASIC cell library successively. The first two percentages are quite misleading as we assume that the embedded multipliers and ALUTs have equal area weight which is not correct. Thus percentage increase in case of ASIC cell library is the most accurate number which is 26%.

The advantage of that design is that we use only one floating point multiplier to implement modal interval multiplier thus decreasing the area and power consumption however, this area and power reduction is balanced by the

loss of speed (The interval operation is half (or one-third) the speed of the normal floating point operation which means we have one interval result each two (or three) clock cycles (in case of pipelining) as shown in tables 3.21, 3.22 and 3.23).

Combined results for the serial and parallel designs are in the following table:

	Cyclone II		Stratix III		Nangate 45nm Cell Library	
	Area Increase %	Clock Frequency (MHz)	Area Increase %	Clock Frequency (MHz)	Area Increase %	Clock Frequency (MHz)
BFP Multiplier	-	128	-	181.5	-	870
Parallel MIBFP Multiplier	148%	120	145.5%	167	128%	870
Serial MIBFP Multiplier	54%	124.5	61.5%	177	26%	870

Table 3.24: Interval multiplier (Combined Results)

We should notice that, the percentage increases are close for the two FPGA devices which are from two different families. The clock frequencies differ from the BFP multiplier for the same device although they should be the same but this is due to variations in the ALTERA CAD tool design rules.

Chapter 4

4 Testing, Comparisons and Future Work

4.1 Testing

4.1.1 Testing Libraries

The Designs are tested using two libraries:

1- INTLAB library which is a classical interval arithmetic MATLAB toolbox [14]. This library is used to test the classical intervals part only. The only deviation from the library is that it doesn't support (-0)

2- IvalDb library which is a C++ modal interval arithmetic library [16]. This library supports handling both proper and improper intervals. One major difference in that library that it doesn't make rounding down and up for the two interval result bounds [16]. Instead, it adds one ULP to the upper bound and subtracts one ULP from the lower bound to guarantee the enclosure of the solution. This leads to a better speed but lower accuracy (wider intervals) as we have a maximum of two ULPs wider interval. The library is modified to properly round interval results. There is another difference in handling infinities. In case of modal interval multiplication many cases gives NAN for the two interval bounds which is not correct.

4.1.2 Test Bench

The same Test Bench method is used in testing the modal interval addition, subtraction and multiplication units. The Testing is divided into two steps. First, Generate the testing vectors then apply input testing vectors to the unit under test (UAT) then compare the output with the output testing vectors. These two steps are explained in more details in the following section.

4.1.2.1 Generate Testing Vectors

The input and output testing vectors are generated using one of the above interval libraries (INTLAB library for classical intervals only or IvalDb library for modal intervals). The MATLAB software tool is used in case of generating the testing vectors using INTLAB while the Microsoft Visual C++ 6.0 software tool is used in case of using the IvalDb library. The input testing vectors are written into a text file (in the hexadecimal format) while the corresponding output vector written into another file (in the hexadecimal format). The following table shows samples of the generated input and output testing vectors respectively for the testing of the multiplication operation. As shown, each entry in the input column represents the two input interval operands (written in the hexadecimal format) and the corresponding entry in the output column represents the actual output after applying those two inputs. Each input test row contains four double floating point numbers adjacent to each other. The first hexadecimal floating point number represents the first operand's first bound, the second number represents the first operand's second bound, the third one represents the second operand's first bound and the last one represents the second operand's second bound. Each output row contains two double floating point numbers written in hexadecimal format. The first number represents the result's first bound and the second one represents the result's second bound.

Input	Output
bfe6131b8bae450b3fe42041085363743fc77be29c123d323fba23cfe3a68848	bfb2083ab042facd3fb070baddb7143a
3fdd1f52644242b3bfe673f888229135c031ef4319daa37fbfe28b85a4442477	3fda0642a2cbcec8bfd0e09805f5c00a
c00002b6ed6c1725404341004edd305fc000058dc268b02bbfe83d8aece84cb8	c05347af35baaab940100845a10c0122
c0256b52b52b52b6bff0a3fab294aa63bfeec9e60acb0f26bfe75723e7989ba1	3fe846595b734f0d40249bc1a3f680a3

Table 4.1: Samples of Input and Output Testing Vectors

4.1.2.2 Running the Test

The test bench was run using the ModelSim simulation tool. The following figure shows block diagram of running the test.

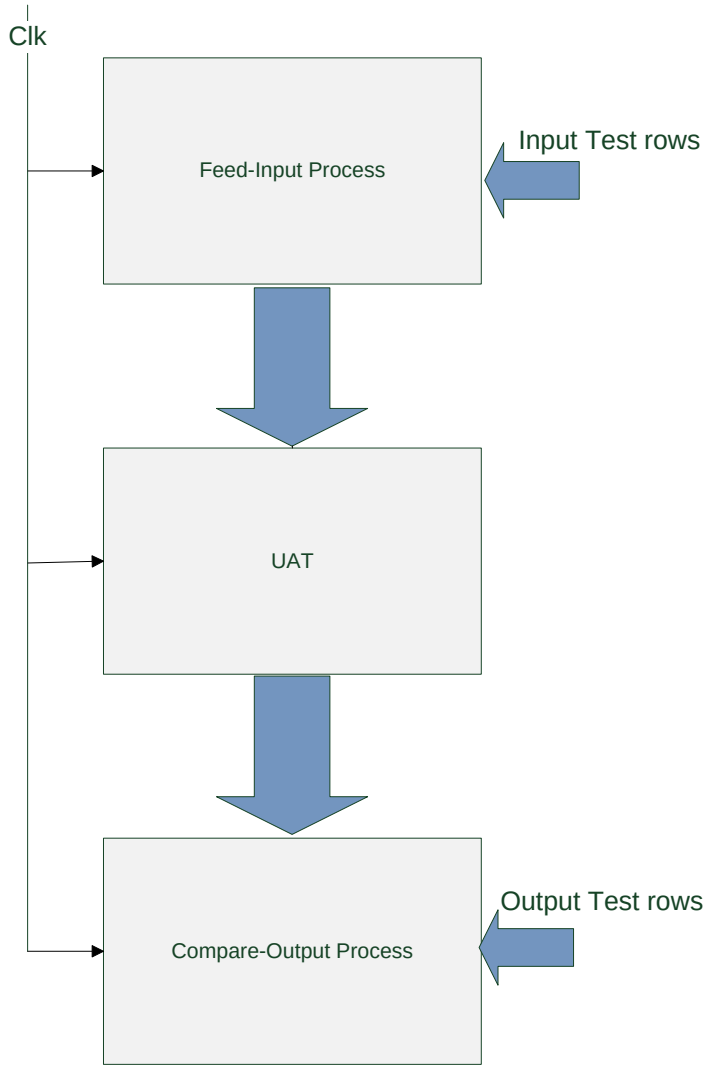

Figure 4.1: Test Bench Block Diagram

The Feed-Input and Compare-Output are running in two parallel clocked processes (VHDL processes). The Feed-Input process reads the testing input operands and feed them into the unit under test (UAT). It feeds input every one clock cycle (if the UAT is a parallel design) or every two clock cycles (if the UAT is a serial design). In case of multiplier, the Feed-Input process should take care of waiting one additional clock cycle when the input operands result in one of the multiplication special cases.

The Compare-Output process starts to compare the output from the UAT with the actual output when the Result_Ready (exists in all modal interval units) flag is logical one. The testing of all the units was done before the logic synthesis (pre-synthesis simulation) and after it (post-synthesis simulation).

4.1.3 Testing coverage

As mentioned before, testing is done using INTLAB (to test the classical part only) and IvalDb to test the more general case (modal and classical intervals). The input testing vector contains of two interval numbers (each one has two floating point numbers which represent the first and second bounds of the interval). As these four input numbers are double floating point numbers, it is impossible to cover all the possible numbers in the test bench. Thus the testing takes two approaches. The first approach is to randomly generate large number of testing inputs. The second approach is to try to divide the floating point domain into different ranges that we can coverage each of them partially. The different ranges are shown in the below table. This allows us to apply testing on the boundaries of the floating point space as well as the normal numbers.

Floating Point Sub-Domain	Description
Norm-Num	Normalized numbers in the normal floating point range
Big-Num	Normalized numbers close to $\pm\infty$
Denorm-Number	Denormalized numbers
± 0	
$\pm\infty$	

Table 4.2: Different floating point ranges

Table 4.3 shows the combinations of the different ranges to generate the input testing vectors.

First Operand		Second Operand	
First Bound	**Second Bound**	**First Bound**	**Second Bound**
Norm-Num	Norm-Num	Norm-Num	Norm-Num
Big-Num	Big-Num	Norm-Num	Norm-Num
Big-Num	Big-Num	Big-Num	Big-Num
Norm-Num	Norm-Num	Big-Num	Big-Num
Denorm-Num	Denorm-Num	Denorm-Num	Denorm-Num
Denorm-Num	Denorm-Num	Norm-Num	Norm-Num
Big-Num	Big-Num	Denorm-Num	Denorm-Num
Norm-Num	Norm-Num	$\pm\infty$	$\pm\infty$
$\pm\infty$	$\pm\infty$	Big-Num	Big-Num
$\pm\infty$	$\pm\infty$	$\pm\infty$	$\pm\infty$
$\pm\infty$	$\pm\infty$	±0	±0
Big-Num	Big-Num	The additive inverse of the first operand's first bound	The additive inverse of the first operand's second bound
Big-Num	Big-Num	The multiplicative inverse of the first operand's first bound	The multiplicative inverse of the first operand's second bound

Table 4.3: Covered ranges in Testing for all units

4.2 Comparison with classical interval counterparts

The major difference between all the previous works (mentioned in Previous Works section) and the proposed implementation is that most of them handled the case of hardware implementation of classical interval arithmetic units (with different approaches). Even for the classical interval multiplication work, most of the previous proposals avoid implementing the special cases of interval multiplications (which need four multiplications instead of two like other cases) and delegate this task to software programs due to its complexity. This is unacceptable (in so many applications) for one of the basic arithmetic

operations like multiplication. Also, none of the classical interval hardware implementations mentioned how infinites as input operands can be handled. Although this case is easy in interval addition/subtraction (handling is like that of the normal floating point adder/subtractor), it is more complex in interval multiplication and needs special circuitry.

To compare the proposed modal interval units with the classical ones, we need to design classical interval units that cover the above two points (handling multiplication special cases and infinities in input). Accordingly, the modal interval adder/subtractor and multiplier are modified to implement the classical interval adder/subtractor and multiplier.

4.2.1 Classical Interval Adder/Subtractor

The only modification that may be needed is to generate an exception flag that indicates that at least one of the interval inputs is a non classical interval (an improper interval). This modification can be delegated to software. In case of implementing the exception flag in hardware, it needs two 64 bit comparators in case of parallel classical interval adder/subtractor (or one 64 bit comparator in case of the serial design) to compare the interval bounds of each input operand. As we noticed, if we implement the improper interval exception flag this leads to an area increase more than that of the modal interval adder/subtractor. Table 4.4 shows different areas (in terms of number of logic elements and number of registers) for different modal/classical and parallel/serial adder/subtractor implementations.

	Serial		Parallel	
	No. of ALUTs	No. of Registers	No. of LEs	No. of Registers
Modal Interval Adder/Sub	1230	997	2358	1716
Classical Interval Adder/Sub	1346	998	2493	1717

Table 4.4: Classical/Modal Add-Sub Area Comparisons (Stratix III)

As shown above, the classical implementations lead to 9.4% utilization increase (of logic elements) in case of serial approach and 5.7% utilization increase (of logic elements) in case of parallel approach.

The Timings won't be affected in pipeline schemes as the delay of the two comparators is embedded into clock cycles that the pre-processing unit takes in both serial and parallel implementations.

As we mentioned, one other possible solution is to leave the improper detection handling to the software. In this case the classical interval adder/subtractor will be the same as its modal counterpart. But there will be timing overhead due to handling of improper interval detection in software.

4.2.2 Classical Interval Multiplier

As we mentioned before, the difference between classical and modal interval units is that we need to check that the input operands are proper intervals, otherwise we generate an exception. This exception can be generated in hardware of by software. If we generate the improper interval exception in hardware, the same modification (done in classical adder/subtractor) will be done in case of multiplication, too. Besides, the logic of selecting the modal interval multiplication case will change as the number of classical multiplication cases is nine cases only. Also, these nine cases don't include cases that lead to Nan results (in case of infinity inputs) which will save another piece of hardware.

Table 4.5 shows different areas (in terms of Logic Elements) for different modal/classical and parallel/serial multiplier implementations. As we can see, in case of hardware support of improper interval detection there is a utilization decrease (in number of logic elements) by 1.3% in the serial implementation while there is a utilization increase (in number of logic elements) by 2.6% in the parallel implementation. On the other hand, in case of leaving improper interval detection to software there is a utilization decrease by 1.3% in the serial implementation (the same percentage as in the hardware support) while

there is utilization decrease by 1.7 in the parallel implementation (on the contrary of the hardware support). These results are totally expected.

	Serial			Parallel		
	No. of ALUTs	No. of Registers	No. of Embedded MULs	No. of LEs	No. of Registers	No. of Embedded MULs
Modal Interval Multiplier	2319	1818	18	3491	2793	36
Classical Interval Multiplier (Hardware exception)	2288	1811	18	3583	2792	36
Classical Interval Multiplier (Software exception)	2287	1810	18	3431	2791	36

Table 4.5: Classical/Modal Multiplier Area Comparisons (Stratix III)

As we said before, in case of hardware support for the improper interval detection at inputs we need comparators to check the two input intervals if the proper or improper intervals. These two comparators are originally present in the serial implementation of the modal interval multiplier (review the Serial Interval Multiplier section), so we don't add them again. But we need to add these two comparators in case of classical parallel implementation. On the other hand, there is lot of logic need to be removed (in both serial and parallel classical implementations) as we handle nine cases in classical interval multiplication instead of sixteen cases in the modal interval multiplication.

The above reasons cause an area decrease in classical serial implementation (with hardware exception) and a reduction in the area increase of the classical parallel implementation (with hardware exception). Apparently, if improper interval detection is left to software, we will have an area decrease in both serial and parallel classical interval multipliers that their modal counterparts.

For Timings, in case of hardware exception the timings won't be affected in pipeline schemes as the delay of the two comparators is embedded into clock cycles that the pre-processing unit takes in both serial and parallel implementations.

4.3 Future Work

The future work need to be done for the current modal interval adder, subtractor and multiplier is to add support for floating point exceptions like overflow, underflow and inexact result exceptions. We should note that the interval results consist of two floating point numbers so we may have two exceptions of the same type in the design or we can just indicate that an exception occurs in the interval result. Another modification can be done in these units is to add the support to operate in two different modes (classical and modal). Thus if the user needs to work with classical intervals only, he will set the Modal flag off so any modal interval input will through and exception.

The previous modifications were to enhance the already done units and they are simple but if we want to make a fully hardware support for the double floating point modal intervals, we need to implement much more units. The first unit that we can think of supporting is the modal interval double floating point divider. Also, there are some important functions like the basic interval elementary functions (trigonometric, exponential and logarithmic functions) and the power functions. All of the previous functions can be built using the corresponding normal floating point units. Another class of operations are the comparison relation operations mentioned in [5]. Although they are built on the normal comparison relations but their definitions are slightly different in modal intervals and have more varieties than those in real numbers system [5], [6]. Another class of operations needed for modal intervals are the midpoint, Infimum, Suprimum and Mode of the interval. The definition of those operations is defined in the "Modal Intervals Building Blocks" section except the midpoint operation which is simply the subtraction of the two bounds divided by two. Two new operations (need to be implemented specifically for intervals) are the intersect and union of two intervals. The definition of both is in [5].

We should note that some of the previous operations (till the moment this thesis were written) are not completely defined in the modal intervals space but they are totally defined in the classical intervals space. Thus they can be implemented with a certain level of support to the modal intervals. The support of all the previous functions in modal intervals will benefit all the users who concern in applications that need accurate results and high speed.

Different topic can be done as a future work is building multiple precision floating point modal interval units. There is no published references mention the need to use multiple precision support for modal intervals but there are many references discuss the need of supporting multiple precision classical intervals [8], [38], [22]. The complex equation which is mentioned in the Introduction chapter (Overview Section) is an example of why we need multiple precision for certain applications. This example shows that the floating point result will be wrong regardless of the precision of the floating point numbers. Unfortunately, when we apply classical interval methods on this example, we have wide interval results for different precisions as shown in Table 4.6 [38].

Precision	**Interval Width**
32-bit	6.3E+30
64-bit	1.1E+22
128-bit	5.1E+03

Table 4.6: Rump's Example: Result Widths using different precision Intervals

We should notice that the more precision we use the tighter interval result we have. Thus using multiple precision intervals will give us the opportunity to have more accurate (tighter) interval results. As the modal intervals support add a slightly more cost to the classical support then it will be better to implement multiple precision modal interval units.

Conclusions

This work introduces for the first time the hardware implementations of the Modal Interval Adder/Subtractor and Modal Interval Multiplier units. It studies two different implementations of these modal interval units using normal double floating point adders/subtrtactors and multipliers respectively. The serial designs represent the execution of the interval operations serially with minimal area increase. On the other hand, the parallel designs make the execution of the interval operations as fast as the execution of normal floating point operations but this comes at the cost of area increase and power consumption. The results shows that the Serial and parallel MIBFP Adder/Subtractors areas are larger than the BFP adder/subtractor area by 16% and 115% respectively .The Serial and parallel MIBFP Multipliers areas are larger than the BFB Multiplier area by 26% and 128% respectively.

Generally speaking, we can say that the hardware serial modal interval multiplier is faster than the different software implementations of modal interval multipliers with a small cost of area increase. Although, the hardware parallel modal interval multiplier is almost double the speed of the serial counterpart, it consumes almost double the area which is a big cost for a unit like the floating point multiplier. According to the modal interval adder/subtractor, only the hardware parallel design makes an improvement over the software implementations but it also costs almost double the area.

It is shown also that the modal interval adder/subtractor and the modal interval multiplier implementations don't have any cost increase than that of their classical counterpart. In fact, it is shown that if we want to add the improper interval detection exception to hardware, this will make the classical units have more areas than those of their classical counterparts (except for the serial modal interval multiplier as explained before).

After all, we can say that only the application nature and the cost are the major benchmarks that determine either the serial approach (smaller but

slower) or the parallel approach (bigger but faster) is suitable.

As the Modal Intervals Analysis is a new branch of Mathematics, there are lots of research points in it, especially in the topic of hardware implementation of the arithmetic units. As mentioned in the "Future Work" section in the previous chapter, there are plenty of Modal Interval arithmetic units that are not implemented in hardware, yet.

References

[1] *D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," ACM Computing Surveys , pp. 5-48, March 1991.*

[2] *R. E.Moore, "Interval Analysis," Prentice Hall Inc., Englewood Cliffs, New Jersey, 1966.*

[3] *R. B. Kearfott, "Interval Computations: Introduction, Uses, and Resources*," Euromath Bulletin, pp.95-112, 1996.

[4] *R. B. Kearfott, "Mainstream Contributions of Interval Computations in Engineering and Scientific Computing*," presented at SCAN 2008 Conference, EL Paso, Texas, Sept., 2008.

[5] *M. A. Ernest Gardenes, "Modal Intervals, Reliable Computing," vol.7, pp.77-111, 2001.*

[6] *N. T. Hayes, "Introduction to Modal Intervals," Prepared for the IEEE 1788 Working Group, 2009.*

[7] *E. Kaucher, "Interval Analysis in the Extended Interval Space IR," Computing Supplementum 2, Springer, Heidelberg, pp. 33–49, 1980.*

[8] *U. Kulish, "Computer Arithmetic and Validity-Theory, Implementation and Applications," Walter de Gruyter GmbH & Co., 2008.*

[9] *R. Gupte, "Interval Arithmetic Logic Unit for DSP and Control Applications*," *MSc., Dept. Elect, Univ. North Carolina, 2006.*

[10] *754-2008 IEEE Standard for Floating-Point Arithmetic,. August 2008. Retrieved from http://ieeexplore.ieee.org/servlet/opac?punumber=4610933*

[11] *P. H. Viñas, "Quantified Real Constraint Solving Using Modal Intervals with Applications to Control*," Ph.D., Dept. Elect, Univ. Girona, 2006.

[12] *Altera Product Catalog, http://www.altera.com/literature/sg/product-catalog.pdf*

[13] *Nangate 45nm Open Cell Library DataBook, http://www.nangate.com*

[14] Classical interval Matlab library (IntLab), *http://www.ti3.tu-harburg.de/rump/intlab/*

[15] Modal interval software library (ivalDb), *http://sites.google.com/site/pauherrero/IVALDB.zip?attredirects=0*

[16] *ivalDb, REFERENCE DOCUMENTATION, http://sites.google.com/site/pauherrero/IVALDB.zip?attredirects=0*

[17] *E. Popova, "On the Efficiency of Interal Multiplication Algorithms," Proceedings of III-rd Int. Conference "Real Numbers and Computers", Paris, pp. 117-132, April 27-29, 1998.*

[18] *E. Popova, S. Markov, "Towards Credible Implementation of Inner Interval Operations," In A. Sydow (Ed.) 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Vol. 2 Numerical Mathematics, pp. 371-376, 1997.*

[19] *"Interval Arithmetic in High Performance Technical Computing," White Paper , Sun Microsystems, Sept. 2002*

[20] *J. E. Stine and M. J. Schulte, "A Combined Interval and Floating Point Multiplier,"* Proceedings of the 8th Great Lakes Symposium on VLSI, Los Alamitos, Feb. 1998.

[21] *J. E. Stine and M. J. Schulte "A Combined Interval and Floating Point Divider,"* Proceedings of the 8th Great Lakes Symposium on VLSI, Los Alamitos, Feb. 1998.

[22] *M. J.Schulte, E. E.Swartzlaner, "Hardware Design and Arithmetic Algorithms for a variable precision, Interval Arithmetic Processor",* Proceedings of the 12th Symposium on Computer Arithmetic, Bath, UK, July. 1995.

[23] *M. J.Schulte, E. E.Swartzlaner, "A family of variable-precision Interval Arithmetic Processors," IEEE Transactions on Computers, vol. 49, pp. 387 - 397, May 2000.*

[24] *R. Kirchner, U. Kulisch, "Hardware Support for Interval Arithmetic," Reliable Computing, Vol. 12, No. 3, 2006.*

[25] *S. Pikoriski, M. Kieffer, L. Lacassagne, D. Etiemble, "Efficient 16-bit Floating-Point Interval Processor for Embedded Systems and Applications," presented at SCAN 2006 Conference, Sept. 2006.*

[26] *UltraSPARC III User's Manual Book, Sun Microsystems, www.sun.com/ultrasparc*

[27] *A. Amaricai, M. Vladutiu, O. Boncalo, "Design of Floating Point Units for Interval Arithmetic,"* In Research in Microelectronics and Electronics, pp.12-15, July 2009.

[28] *Interval Arithmetic,* http://en.wikipedia.org/wiki/Interval_arithmetic

[29] *SIGLA/X Group (Calm R., Estela M.R., et. al.), "Ground Construction of Modal Intervals," Proc. of MISC'99, University of Girona, Spain.*

[30] *SIGLA/X Group (Calm R., Estela M.R., et. al.), "Interpretability and Optimality of Rational Functions," Proc. of MISC'99, University of Girona, Spain.*

[31] *SIGLA/X Group (Calm R., Estela M.R., et. al.), "Semantic and Rational Extensions of Real Continuous Functions," Proc. of MISC'99, University of Girona, Spain.*

[32] *T. Sunaga, Theory of interval algebra and its application to numerical analysis, Research Association of Applied Geometry (RAAG) Memoirs, Vol. 2, pp.29-46, 1958.*

[33] *M. Warmus, Calculus of Approximations, Bull. Acad. Polon. Sci., Cl. III, Vol. IV, No. 5, pp. 253–259, 1956.*

[34] *M. Warmus, Approximations and Inequalities in the Calculus of Approximations. Classification of Approximate Numbers, Bull. Acad. Polon. Sci. Ser. Math. Astr. Phys., Vol. IX, No. 4, pp. 241–245, 1961.*

[35] *S.F. Oberman "Floating Point Arithmetic Unit Including an Efficient Close Data Path" US Patent 6094668, Advanced Micro Devices, 2000.*

[36] *P.M. Seidel, G. Even "Delay-Optimized Implementation of IEEE Floating Point Addition" IEEE Transaction on Computers, Vol. 53, No.2 , pp. 97-113, 2004.*

[37] *S.M. Rump "Verification methods: Rigorous results using floating-point arithmetic," Acta Numerica, pp.287-449, 2010.*

[38] *E. Loh, W. Walaster "Rump's Example Revisited," Reliable Computing, pp.245-248, 2002.*

[39] *Y. Wang "Semantic Tolerance Modeling based on Modal Interval Analysis," proceedings of NSF Workshop on Reliable Engineering Computing (REC'06), pp.293-318, 2006*

[40] *OpenCores IEEE-754 compliant double-precision Floating Point Unit, http://opencores.org/project,fpu_double*