Cairo University,
Faculty of Science,
Mathematics Department
Computer Science Division.

INTERVAL ARITHMETIC

Accurate Self-validating Arithmetic for Digital Computing

A Graduate Report in Computer Arithmetic

Hend Dawood

July 2007

Supervisor: Dr. Hossam A. H. Fahmy

INTERVAL ARITHMETIC

Accurate Self-validating Arithmetic for Digital Computing

Hend Dawood

Cairo University, Faculty of Science, Department of Mathematics, Computer Science Division
Email: hend.dawood@gmail.com

Graduate Report in Computer Arithmetic
Hend Dawood, July 2007

Abstract

Interval arithmetic (range arithmetic) is a broad field in which rigorous mathematics is associated with scientific computing. It is an arithmetic defined on sets of intervals rather than sets of real numbers. The connection between computing and mathematics provided by intervals makes it possible to solve problems that can't be efficiently solved using traditional floating point arithmetic. Today, the interval methods are becoming rapidly popular as a perspective weapon against round-off errors. A number of researchers worldwide produced a voluminous literature on the subject. This report introduces the theoretical aspects of interval arithmetic, as well as some of its computational and scientific applications. Also, we introduce the hardware implementations of a 4-by-4 bit multiplier and an interval squaring circuit.

Keywords: Interval Arithmetic, Range Arithmetic, Reliable Computing, Round-off Errors, Self-validating Arithmetic, 4-by-4 Multiplier, Interval Squaring Circuit.

Table of Contents

1. Prologue: History of Interval Arithmetic

Interval arithmetic is an arithmetic defined on sets of intervals, rather than sets of real numbers. A form of interval arithmetic perhaps first appeared in 1924 by J. C. Burkill in his paper "*Functions of Intervals*" [1] and in 1931 by R. C. Young in his paper "*The Algebra of Many-Valued Quantities*" [2] that gives rules for calculating with intervals and other sets of real numbers, then later in 1958 by T. Sunaga in his book"*Theory of an Interval Algebra and its Application to Numerical Analysis*" [3]. Modern developments of interval arithmetic began in 1959 with R. E. Moore's technical report "*Automatic Error Analysis in Digital Computation*" in which he developed a number system and an arithmetic dealing with closed real intervals. He called the numbers "Range Numbers" and the arithmetic "Range Arithmetic" [4] to be the first synonyms of "Interval Numbers" and "Interval Arithmetic". Then later in 1962, Moore developed a theory for exact or infinite precision interval arithmetic in his very influential dissertation titled "*Interval Arithmetic and Automatic Error Analysis in Digital Computing*" [5] in which he used a modified digital (rounded) interval arithmetic as the basis for automatic analysis of total error in a digital computation. Since then, thousands of research papers and numerous books have appeared on the subject.

Today the interval methods are becoming rapidly popular as a perspective weapon against round-off errors. There is an increasing amount of software support for interval computations. Implementations of interval arithmetic are available both as specialized programming languages and as libraries that can be linked to a program written in a standard language. There are even interval spreadsheet programs and interval calculations [6].

One thing the interval community has been ardently seeking- so far without success- is support for interval algorithms in standard computer hardware. Efforts are being paid to design interval arithmetic units, but manufacturers have not chosen to provide chips with built-in facilities for interval computations, which are technologically feasible, analogous to the provided built-in floating point arithmetic facilities. Chipmakers are still wary of devoting resources to facilities no one might use [7].

In the following sections, we introduce the theory of interval arithmetic and a number of its applications for digital computing, as well as some widespread applications for other scientific and engineering fields. Also, we provide a discussion of some implementations of interval arithmetic at hardware level.

2. Interval Arithmetic and Interval Numbers

Interval arithmetic (range arithmetic) was proposed in its modern form by Moore [8] as a tool for bounding round off errors in numerical computations. Interval arithmetic specifies a precise method for performing arithmetic operations on intervals (interval numbers). In the interval number system, each interval number represents some fixed real number between the lower and upper endpoints of the interval. So, an interval arithmetic operation produces two values for each result. The two values correspond to the lower and upper endpoints of the resulting interval, such that the true result is guaranteed to lie on this interval. The accuracy of the result is indicated by the width of the resulting interval (i.e., the distance between the two endpoints).
In what follows, we discuss the theoretical aspects of real, complex, and digital (rounded) interval arithmetic.

2.1. Real Interval Arithmetic:

Where classical arithmetic defines operations on individual real numbers, real interval arithmetic is based on arithmetic within the set of closed intervals of real numbers. The concept is simple: real numbers are not represented as discrete values, but as ranges (intervals) in which the actual (i.e., correct) value is known to reside. The interval width indicates the maximum possible error.
For example: The Archimedes' constant, π , can be represented as the interval number:

$$\begin{aligned}\Pi &= [\pi_l, \pi_u] \\ &= \left[314\times10^{-2}, 315\times10^{-2}\right].\end{aligned}$$

Since,

$$314\times10^{-2} \le \pi \le 315\times10^{-2}.$$

And the maximum possible error is indicated by the width of the interval number Π ,

$$\begin{aligned}Error &= width(\Pi) = \pi_u - \pi_l \\ &= 315\times10^{-2} - 314\times10^{-2} \\ &= 10^{-2}.\end{aligned}$$

Operations involving Π are not "aware" that Π represents π [9]. The operations only assume that Π represents some fixed real number between 314×10^{-2} and 315×10^{-2} .
Many real interval arithmetic operations are similar to "traditional" arithmetic operations, except that they compute the upper and lower bounds rather than just a single number. In the following discussion of the operations and properties of real interval arithmetic, we use the following notations and definitions:

- Intervals (interval numbers) are denoted by capital letters and real numbers are denoted by small letters. The lower and upper interval endpoints of an interval X are denoted by x_l and x_u , respectively.

- A closed interval $X = [x_l, x_u]$ consists of set of all real numbers x for which $x_l \le x \le x_u$.
- The set of all closed intervals is denoted by S. S is necessarily infinite [10].
- A real number x is equivalent to the degenerate interval $[x, x]$. The set of all degenerate closed intervals of the form $[x, x]$ (i.e., of zero width) is denoted by S^D. S^D is a subset of S, necessarily infinite, and isomorphic to the field of real numbers [11].
- The set of all intervals that do not contain the real number 0 is denoted by S^*. S^* is a subset of S and necessarily infinite.
- The equality and inequality relations for two intervals $[x_l, x_u]$ and $[y_l, y_u]$ are defined as follows [12]:

$$\begin{aligned} [x_l, x_u] &= [y_l, y_u] \quad iff \quad x_l = y_l \text{ and } x_u = y_u \\ [x_l, x_u] &< [y_l, y_u] \quad iff \quad x_u < y_l \\ [x_l, x_u] &> [y_l, y_u] \quad iff \quad x_l > y_u \end{aligned} \tag{2.1}$$

2.1.1. Scalar Interval Operations:

Several operations decompose intervals into scalar values. These include infimum (inf), supremum (sup), magnitude (mag), width, and midpoint (mid) of an interval. The scalar decomposition operations of an interval $X = [x_l, x_u]$ are shown in Table 2-1.

Scalar Operation	Examples
$\inf(X) = x_l$	$\inf([1,3]) = 1$
$\sup(X) = x_u$	$\sup([1,3]) = 3$
$\text{mag}(X) = \max(\lvert x_l \rvert, \lvert x_u \rvert)$	$\text{mag}([-2,-1]) = 2$
$\text{width}(X) = x_u - x_l$	$\text{width}([4.2,4.4]) = 0.2$
$\text{mid}(X) = (x_l + x_u)/2$	$\text{mid}([4.2,4.4]) = 4.3$

Table 2-1: Scalar Interval Operations

2.1.2. Binary Interval Operations:

Six binary operations are defined for intervals; the four basic arithmetic operations (i.e. addition, subtraction, multiplication, and division), plus the intersection and hull. For two interval numbers $X = [x_l, x_u]$ and $Y = [y_l, y_u]$, the binary interval operations are defined as follows:

- Addition and Subtraction: interval addition and subtraction are defined as:

$$X + Y = [x_l + y_l, x_u + y_u] \tag{2.2}$$

$$X - Y = [x_l, x_u] - [y_l, y_u]$$
$$= [x_l, x_u] + [-y_u, -y_l] \quad (2.3)$$
$$= [x_l - y_u, x_u - y_l]$$

For example,

$$[4.2, 4.4] + [3.4, 3.5] = [7.6, 7.9]$$
$$[4.2, 4.4] - [3.4, 3.5] = [0.7, 1.0]$$

- Multiplication: interval multiplication is defined as:

$$X \bullet Y = [\min(x_l y_l, x_l y_u, x_u y_l, x_u y_u), \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)] \quad (2.4)$$

For example,

$$[4.2, 4.4] \bullet [3.4, 3.5] = [14.28, 15.40] \simeq [14, 16]$$

Here, the resulting interval number is outward rounded to two decimal digits.

- Division: if $Y \in S^*$ (i.e. Y does not contain 0), then, interval division can be defined as, [13]

$$X / Y = [x_l, x_u] / [y_l, y_u]$$
$$= [x_l, x_u] \bullet [1/y_u, 1/y_l] \quad (2.5)$$

- Intersection and hull: the intersection and hull (union) of two intervals X and Y are defined as [14] [15]:

$$\text{intersect}(X, Y) = X \cap Y$$
$$= \{x \mid x \in [x_l, x_u] \wedge x \in [y_l, y_u]\} \quad (2.6)$$
$$= [\max(x_l, y_l), \min(x_u, y_u)]$$

$$hull(X, Y) = X \cup Y$$
$$= \{x \mid x \in [x_l, x_u] \vee x \in [y_l, y_u]\} \quad (2.7)$$
$$= [\min(x_l, y_l), \max(x_u, y_u)]$$

Interval intersection returns $\varnothing$ if $\max(x_l, y_l) > \min(x_u, y_u)$.

For example,

$$\text{intersect}([4.2, 4.4], [3.4, 3.5]) = \varnothing$$
$$\text{hull}([4.2, 4.4], [3.4, 3.5]) = [3.4, 4.4]$$

2.1.3. Unary Interval Operations:

Common unary interval operations include absolute value, negation, square, and square root. These unary operations for $X = [x_l, x_u]$ are shown in Table 2-2.

Unary Operation	Examples
Absolute value: $\lvert X \rvert = [\min(\lvert x_l \rvert, \lvert x_u \rvert), \max(\lvert x_l \rvert, \lvert x_u \rvert)]$; *if* $X \in S^*$ $\lvert X \rvert = [0, \max(\lvert x_l \rvert, \lvert x_u \rvert)]$; *if* $X \notin S^*$	$\lvert [1,3] \rvert = [1,3]$ $\lvert [-3,-1] \rvert = [1,3]$ $\lvert [-3,1] \rvert = [0,3]$

Negation: $-X = [-x_u, -x_l]$	$-[4,5] = [-5,4]$ $-[-3,1] = [-1,3]$
Square: $X^2 = [\min(x_l^2, x_u^2), \max(x_l^2, x_u^2)]$	$[2,3]^2 = [4,9]$ $[-3,-2]^2 = [4,9]$ $[-3,2]^2 = [4,9]$ $[-2,3]^2 = [4,9]$
Square root: $\sqrt{X} = [\sqrt{x_l}, \sqrt{x_u}]\ ; if\ \ X\ is + ve$ $\sqrt{X} = [Nan, Nan]\ ; if\ \ X\ is - ve$ $\sqrt{X} = [0, \sqrt{x_u}]\ ; if\ \ X \notin S^*$	$\sqrt{([1,4])} = [1,2]$ $\sqrt{([-4,-1])} = [Nan, Nan]$ $\sqrt{([-1,1])} = [0,1]$

Table 2-2: Unary Interval Operations

2.1.4. Properties of Real Interval Numbers:

In this section we discuss briefly the basic properties of real interval numbers. These properties are particularly important for the study of interval variables, interval expressions, and interval functions. For all elements of the set of interval numbers S, the following properties are true [16] [17] [18]:

- Closure for addition: for every pair of elements X and Y in S, there exists a unique element,
$$Z = X + Y \text{ in } S.$$
- Closure for multiplication: if X and Y are any pair of elements in S, there exists a unique element,
$$Z = X \bullet Y \text{ in } S.$$
- Associativity for addition: for all elements X, Y, Z in S,
$$X + (Y + Z) = (X + Y) + Z$$
- Associativity for multiplication: for all elements X, Y, Z in S,
$$X \bullet (Y \bullet Z) = (X \bullet Y) \bullet Z$$
- Commutativity for addition: for every pair of elements X and Y in S, the relation,
$$X + Y = Y + X \text{ holds.}$$
- Commutativity for multiplication: if X and Y are any pair of elements in S, then,
$$X \bullet Y = Y \bullet X$$
- Identity for addition: the closed interval $[0,0]$ is both a left and right identity for addition, i.e.,
$$[0,0] + X = X + [0,0] = X$$

- Identity for multiplication: the closed interval [1,1] is both a left and right identity for multiplication, i.e.,
$$[1,1]\bullet X = X \bullet [1,1] = X$$
- Reflexive law: $X = X$
- Symmetric law: if $X = Y$, then, $Y = X$
- Additive inverses: additive inverses don't exist in the set of closed intervals S, except for the subset S^D.
- Multiplicative inverses: multiplicative inverses don't exist in S, except for $S^D - \{[0,0]\}$.
- For all elements of S^D, we have,
$$[x,x]+[y,y]=[x+y,x+y]$$
$$[x,x]-[y,y]=[x-y,x-y]$$
$$[x,x]\bullet[y,y]=[xy,xy]$$
$$[x,x]/[y,y]=[x/y,x/y],\ y\neq 0.$$
So that interval arithmetic includes real arithmetic, identifying the interval $[x,x]$ with the real number x.
- Cancellation law for addition: for X, Y, Z in S, if:
$$X+Y=X+Z$$
Then, $Y = Z$.
- Cancellation law for multiplication doesn't hold in interval arithmetic. That is, for the intervals X, Y, Z in S, if:
$$X \bullet Y = X \bullet Z$$
We can't conclude that: $Y = Z$.
- Addition and Multiplication of a real number c and a closed interval X is equivalent to the addition and multiplication of the degenerate interval $[c,c]$ and the closed interval X, i.e.,
$$c + X = [c,c] + X$$
$$c \bullet X = [c,c] \bullet X$$
- The distributive law doesn't hold in interval arithmetic, except when the common factor is an element in S^D. For the intervals X, Y in S and A in S^D,
$$A \bullet (X+Y) = A \bullet X + A \bullet Y$$
Due to the isomorphism between the real numbers and the set S^D, we can deduce, for any real number a, that,
$$a \bullet (X+Y) = a \bullet X + a \bullet Y$$
- The elements of S are partially ordered by set inclusion. In fact,
$$[a,b] \subset [c,d] \ \textit{iff} \ \ c \le a \le b \le d.$$
- For any closed interval $X \in S$, and any integer $n \ge 0$,
$$X^n = X \bullet X \bullet ... \bullet X \ \text{(n times)}$$
And for $n = 0$,
$$X^0 = [1,1]$$

2.1.5. Rational Interval Expressions [19]:

A rational interval expression

$$F([x_1,x_2],[x_3,x_4],...,[x_{n-1},x_n])$$

Is a finite combination of closed interval variables,

$$[x_1,x_2],[x_3,x_4],...,[x_{n-1},x_n]$$

And a finite set of constant closed intervals of the form $[a,b]$ in an expression with interval arithmetic operations.

A rational interval form is usually not representable as a quotient of two polynomials. That is, we can't say:

$$[x_1,x_2]+\frac{[1,1]}{[x_1,x_2]}=\frac{[x_1,x_2]^2+[1,1]}{[x_1,x_2]},$$

unless $x_1 = x_2$.

Since interval arithmetic operations are monotonic inclusive, if:

$$[x_1',x_2']\subset[x_1,x_2],\ [x_3',x_4']\subset[x_3,x_4],...,\ [x_{n-1}',x_n']\subset[x_{n-1},x_n].$$

And if:

$$F([x_1,x_2],[x_3,x_4],...,[x_{n-1},x_n])$$

is a rational interval expression, then:

$$F([x_1',x_2'],[x_3',x_4'],...,[x_{n-1}',x_n'])\subset F([x_1,x_2],[x_3,x_4],...,[x_{n-1},x_n]).$$

2.2. Complex Interval Arithmetic:

Instead of limiting the application of interval numbers to the measure of uncertainty in real numbers, we could use interval numbers to determine a region of uncertainty in computing with complex numbers. So, here, we wish to develop complex interval numbers. [20]

Definitions and operations:

Definition (1): A complex interval number Z is defined by,

$$Z = A + iB = (A,B)$$

Where the capital letters A and B refer to real interval numbers, and i refer to the complex interval number $([0,0],[1,1])$.

So, if $A=[a,b]$ and $B=[c,d]$, then,

$$Z = [a,b]+[c,d]\ i$$
$$= \{x + y\ i \mid a \le x \le b\ ,\ c \le y \le d\}.$$

Definition (2): if $Z=(A,B)$ is a complex interval number, then the *conjugate* of Z denoted by $\bar{Z}$ is defined by:

$$\bar{Z} = (A,-B)$$

Definition (3): if $Z=(A,B)$ is a complex interval number, then the *negative* of Z denoted by $-Z$ is defined by:

$$-Z = (-A,-B)$$

Now, we define the sum, product, and quotient of two complex interval numbers [21].

Definition (4): the sum of two complex interval numbers (A, A') and (B, B') is the complex interval number $(A+B, A'+B')$, with the alternative notation,

$$([a,b]+[c,d]\,i)+([e,f]+[g,h]\,i)=[a+e,b+f]+[c+g,d+h]\,i.$$

Definition (5): the product of two complex interval numbers (A, A') and (B, B') is the complex interval number $(AB-A'B', AB'+A'B)$.

Then the quotient is defined as follows:

Definition (6): if (A, A') and (B, B') are two complex interval numbers and $0 \notin (B, B')$, their quotient, $(A, A') \big/ (B, B')$, is defined as the complex interval number (C, C') where,

$$\begin{aligned}(C,C') &= \frac{(A,A')}{(B,B')} \\ &= \frac{(A,A')\bullet(B,-B')}{(B,B')\bullet(B,-B')} \\ &= \frac{(A,A')\bullet(B,-B')}{(B^2-B'^2,0)} \\ &= \frac{(AB+A'B', A'B-AB')}{(B^2-B'^2,0)} \\ &= \left(\frac{AB+A'B'}{B^2-B'^2}, \frac{A'B-AB'}{B^2-B'^2}\right)\end{aligned}$$

providing $0 \notin B^2-B'^2$.

2.3. Digital (Rounded) Interval Arithmetic:

If the end points of interval numbers are restricted to lie in a set of digitally representable numbers, the resulting interval numbers are called *digital interval numbers*.

For example, the range of the expression $[0.123, 0.456]+[0.0116, 0.0214]$ is $[0.1346, 0.4774]$. But this would be rounded to $[0.135, 0.477]$ with three digit decimal arithmetic and rounding to nearest, and $[0.1346, 0.4774] \not\subset [0.135, 0.477]$. Nonetheless, with directed rounding, such bounds can be computed rigorously.

In particular, if instead of rounding to nearest, the lower bound of the interval is rounded down to the largest machine number less than the exact result and the upper bound is rounded up to the smallest machine number greater than the actual result, then the computed interval necessarily contains the exact range.

In our example, the result would be

$$[0.134, 0.475], \text{ and } [0.1346, 0.4774] \subset [0.134, 0.475].$$

We conclude that, with directed rounding, machine interval arithmetic can be defined, such that a direct digital range computation will produce as results sets of real intervals with digital numbers as end points. In these intervals will lie the exact results of the corresponding real arithmetic computations.

3. Computational Applications of Interval Arithmetic

Interval arithmetic provides several elementary and powerful tools such as bounding the ranges of functions (for example, bounds on the range of an objective function are extremely useful in global optimization algorithms). Also, with wide use in scientific computing, is bounding the error term in Taylor's theorem. Finally, in some calculations, interval arithmetic (with directed roundings) can be used to bound the effects of round-off error. [22]

In this section we will explain in some detail the interval computation of some elementary functions, how interval arithmetic is used to express the remainder term in Taylor's theorem as an interval, and how interval arithmetic is applied to evaluate definite integrals.

3.1. Interval Computation of Elementary Functions:

Interval arithmetic could be used to obtain bounds on the range of a function that could be evaluated as a sequence of the four elementary operations. In this section we explain how we perform this on some examples of elementary functions.

In the calculation to follow, we use the following notations, definitions and propositions [23]:

- Let $A=[a_1,a_2]$ and $B=[b_1,b_2]$ be two interval numbers where:
  1. $(-1)A=[-1,-1]\ A=[-a_2,-a_1]$ will be denoted by $-A$.
  2. $A\vee B=[\min\{a_1,b_1\},\max\{a_2,b_2\}]$.
  3. $A\odot B=A+(-B)=[a_1-b_2,a_2-b_1]$.
  4. $A\oplus B=A-(-B)=[a_1+b_2\vee a_2+b_1]$.
  5. $A^0=1,\ A^n=A\times A\times A\times\ldots\times A$ n times.

- Assume that f is continuous and monotone (c.m.) function on $D\in IR$ [the set of all real intervals] , for any $X=[x_1,x_2]\subset D$ the set $\{f(x):x\in X\}$ is an interval which is easy to compute:

  $$\{f(x):x\in X\}=[f(x_1)\vee f(x_2)]$$

  This interval will be denoted by $F(X)$, such that:

  $$F:D\to IR$$

- Consider two functions f,g which are c.m. on D. We shall distinguish between the following two situations:
  i. Both functions f,g are monotone increasing on D or both are monotone decreasing on D; we shall then say that f and g are **equimonotone** (e.m.) on D.
  ii. One of the functions f,g is monotone increasing on D and the other is monotone decreasing on D; in this case we say that f and g are **differently monotone** (d.m.) on D.
- Now, let's state the following two propositions:

Proposition (1): the functions f, g and $f+g$ are c.m. on D. Then for every interval $X \subset D$:

$$\{f(x)+g(x) : x \in X\} = \begin{cases} F(X)+G(X) & ; if\ f\ and\ g\ are\ e.m.on\ D, \\ F(X) \oplus G(X) & ; if\ f\ and\ g\ are\ d.m.on\ D. \end{cases}$$

Proposition (2): the functions f, g and $f-g$ are c.m. on D. Then for every interval $X \subset D$:

$$\{f(x)-g(x) : x \in X\} = \begin{cases} F(X)-G(X) & ; if\ f\ and\ g\ are\ e.m.on\ D, \\ F(X) \odot G(X) & ; if\ f\ and\ g\ are\ d.m.on\ D. \end{cases}$$

Let's now use the interval arithmetic to evaluate some elementary functions [24].

(1) Consider the **cosine** and **sine** functions, it is sufficient to obtain interval formulas that are valid for $0 \le X \le \frac{\pi}{4}$.

The cosine function:

$$\cos x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{(2k)!}$$

We notice that all partial sums are monotone decreasing functions on $[0, \sqrt{6}]$ and by propositions (1) and (2), we obtain for the range $\cos X = \{\cos x : x \in X\}$:

$$\cos X = 1 \odot \frac{X^2}{2!} \oplus \frac{X^4}{4!} \odot \frac{X^6}{6!} \oplus \frac{X^8}{8!} \odot \ldots,$$

for $X \subset [0, \sqrt{6}]$.

Since $\sqrt{6} > \frac{\pi}{4}$; the last formula can be used for the computation of $\cos X$ for any X. Also, since the functions $\sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!}$ are monotone increasing on $[-\sqrt{6}, 0]$ for every n, then the previous formula holds true for $-\sqrt{6} \le X \le 0$ as well.

The sine function:

$$\sin X = \{\sin x : x \in X\}$$

Since the partial sums in the Taylor expansion of $\sin x$ are monotone increasing functions on $[-\sqrt{2}, \sqrt{2}]$ for every fixed number n of the summands, we get:

$$\sin X = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} - \ldots,$$

for $-\sqrt{2} \le X \le \sqrt{2}$.

(2) Consider the **logarithmic** function $\ln(1+x)$. For $x \in (-1, 1]$ we have:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots,$$

Since all partial sums are monotone decreasing on $(-1,1]$ we obtain for the range $\{\ln(1+x) : x \in X\} = \ln(1+X)$:

$$\ln(1+X) = \begin{cases} X - \frac{X^2}{2} + \frac{X^3}{3} - \frac{X^4}{4} + \ldots ; for\ 0 \le X \le 1, \\ X \odot \frac{X^2}{2} + \frac{X^3}{3} \odot \frac{X^4}{4} + \ldots ; for\ -1 < X \le 0. \end{cases}$$

3.2. Taylor's Series [25]:

We shall discuss Taylor's series since it is often used on digital computers for approximating functions, and shall place our emphasis on the remainder term of the series.

Taylor's theorem: let a function $f(x)$ and its first n derivatives ($n \ge 0$) be continuous in a closed interval containing $x = a$, and let x be any point in this interval. Then:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \ldots + \frac{(x-a)^{n-1}}{(n-1)!}f^{(n-1)}(a) + R_n, \quad (3.1)$$

Where R_n (the remainder) is given by:

$$R_n = \frac{1}{(n-1)!}\int_a^x (x-t)^{n-1} f^{(n)}(t)\,dt.$$

This form of the remainder is usually rather difficult to estimate in a numerical problem. Therefore we will develop Lagrange's form of the remainder:

$$R_n = \frac{(x-a)^n}{n!} f^{(n)}(\xi)\ ,\ a < \xi < x$$

which has the advantage of simplicity and, is more amenable to interval arithmetic.

If we set $a = 0$ in (3.1), we get the well-known special case, called Maclaurin's series with remainder:

$$f(x) = f(0) + xf'(0) + \frac{x^2}{2!}f''(0) + \ldots + \frac{x^{n-1}}{(n-1)!}f^{(n-1)}(0) + R_n,$$

With,

$$R_n = \frac{x^n}{n!} f^{(n)}(\xi)\ ,\ 0 < \xi < x\,.$$

The remainder term as an interval:

Our objective here is to obtain a closed interval which will contain the exact value of the remainder R_n.

We have:

$$R_n = \frac{(x-a)^n}{n!} f^{(n)}(\xi)\ ,\ a < \xi < x\ ,$$

Then the exact value is dependent upon ξ. We will build a closed interval which will contain all possible values of R_n, for $a < \xi < x$.
For simplicity, we make the following notational definitions:

Definition (1): $R_{n,a} = \frac{(x-a)^n}{n!} f^{(n)}(a)$.

Definition (2): $R_{n,x} = \frac{(x-a)^n}{n!} f^{(n)}(x)$.

It is clear that either $R_{n,a} \le R_n \le R_{n,x}$ or $R_{n,x} \le R_n \le R_{n,a}$.
Now, we define the closed interval which contains R_n.

Definition (3): $R_{n,i} = [\min(R_{n,a}, R_{n,x}), \max(R_{n,a}, R_{n,x})]$.

Theorem: The closed interval, $R_{n,i}$, contains R_n.

Taylor's series Interval Algorithm:
If we modify the usual form of the Taylor series (3.1), to make use of the closed interval $R_{n,i}$ (which contains R_n), we get:

$$f(x) \subset \sum_{j=0}^{n-1} \frac{1}{j!}(x-a)^j f^{(j)}(a) + R_{n,i}. \qquad (3.2)$$

When $f(x)$ is any real function which can be computed by Taylor's series, and when the n^{th} derivative is either increasing or decreasing monotonically between a and x, then the first n terms can be computed in the usual manner and then added to the closed interval $R_{n,i}$.

In practice, it is often necessary to start a computation with an inexact value of a parameter x, say, $x \pm \varepsilon$.

e.g., $f(x) = e^x$ with $x = 1 \pm 0.1$ or $f(x) = \sin x$ with $x = 0.25 \pm 0.001$.

We will now develop a procedure to obtain a closed interval in which the exact value of the function is known to lie.
If x is a real number, so are $x + \varepsilon$ and $x - \varepsilon$. Then, $f(x+\varepsilon)$ and $f(x-\varepsilon)$ can both be calculated by Taylor's series (3.1) or by (3.2), i.e., Taylor's series with a closed interval remainder $R_{n,i}$, assuming increasing or decreasing monotonicity of the n^{th} derivative between a and x.
Let us define the following:

Definition (4): $f(x-\varepsilon) \subset \sum_{j=0}^{n-1} \frac{1}{j!}(x-\varepsilon-a)^j f^{(j)}(a) + R_{n,\alpha}$, where, $R_{n,\alpha} = R_{n,i}$ with x replaced by $x - \varepsilon$, i.e.,

$$R_{n,\alpha} = [\min(R_{n,a}, R_{n,x-\varepsilon}), \max(R_{n,a}, R_{n,x-\varepsilon})]$$

Where the condition on the remainder term is now: $a < \xi < x - \varepsilon$.

Definition (5): let α be as follows:

$$\alpha = \sum_{j=0}^{n-1} \frac{1}{j!}(x-\varepsilon-a)^j f^{(j)}(a).$$

Then, $f(x-\varepsilon) \subset \alpha + R_{n,\alpha}$ or, $f(x-\varepsilon) \subset [\alpha + (R_{n,\alpha})_{\min}, \alpha + (R_{n,\alpha})_{\max}]$, where $(R_{n,\alpha})_{\min}$ and $(R_{n,\alpha})_{\max}$ are the left and right ends of the closed interval $R_{n,\alpha}$, respectively.

For $f(x+\varepsilon)$ we can also make the corresponding definitions using β instead of α .i.e., $R_{n,\beta} = R_{n,i}$ with x replaced by $x+\varepsilon$ and $\beta = \sum_{j=0}^{n-1} \frac{1}{j!}(x+\varepsilon-a)^j f^{(j)}(a)$.

Then, $f(x+\varepsilon) \subset \beta + R_{n,\beta}$ or, $f(x+\varepsilon) \subset [\beta + (R_{n,\beta})_{\min}, \beta + (R_{n,\beta})_{\max}]$.

Now, the following is an algorithm for computing a closed interval which contains all values of the function $f(x \pm \varepsilon)$, and which also contains the upper and lower error bounds.

Algorithm: If $f(x)$ meets the conditions of Taylor's theorem, and if the n^{th} derivative is monotonically increasing or decreasing between a and x, then for $f(x \pm \varepsilon)$ we have the Taylor series interval algorithm:

$$f(x \pm \varepsilon) \subset [\min(\alpha + (R_{n,\alpha})_{\min}, \beta + (R_{n,\beta})_{\min}), \max(\alpha + (R_{n,\alpha})_{\max}, \beta + (R_{n,\beta})_{\max})]. \quad (3.3)$$

We note that if $\varepsilon = 0$, then $\alpha = \beta$ and (3.3) becomes (3.2).

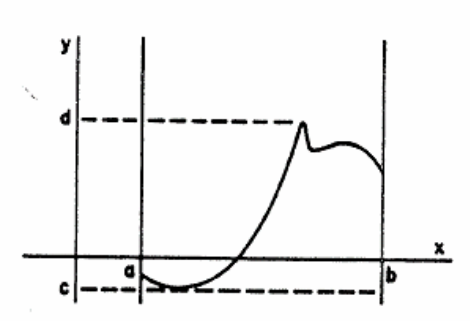
3.3. Evaluation of Definite Integrals [26]*:*

In this section we explain how interval arithmetic is applied to evaluate definite integrals.

Let $f(x)$ be a real valued function such that $\int_a^b f(x)\,dx$ exists and such that the associated interval valued function $F(X)$ is defined for $X = [a,b]$.

Let $Y = [c,d] = F(X)$, and let $y = f(x)$. If x takes on any value such that $a \le x \le b$, we deduce directly that $y \in Y$, and hence we conclude that $Y = [c,d]$ includes all values that $f(x)$ may take on in the interval $[a,b]$. In particular we note that c is a lower bound and d is an upper bound for f in $[a,b]$.

From the following figure we see that $d(b-a)$ and $c(b-a)$ are an upper and lower bound, respectively, on $\int_a^b f(x)\,dx$.



Selecting intermediate points $a < a_1 < a_2 < \ldots < a_n < b$ leads to a computing algorithm since,

$$\int_a^b f(x)\,dx = \int_a^{a_1} f(x)\,dx + \int_{a_1}^{a_2} f(x)\,dx + \ldots + \int_{a_n}^{b} f(x)\,dx.$$

4. Hardware Implementation of Interval Arithmetic (Interval Arithmetic Units)

Interval arithmetic provides an efficient method for monitoring and controlling errors in numerical calculations and can be used to solve problems that can't be efficiently solved with floating-point arithmetic. However, existing software packages for interval arithmetic are often too slow for numerically intensive calculations. While conventional floating point arithmetic is provided by fast hardware, interval arithmetic is simulated with software routines based on integer arithmetic. Therefore, the hardware design for interval arithmetic can provide a significant performance improvement over software implementations of interval arithmetic [27]. In this section, we develop an interval adder and an interval Squaring ciruit.

4.1. Interval Adder:

Interval addition is very simple to perform by adding the interval bounds and rounding the results appropriately:

$$[a,b]+[c,d]=[\nabla(a+c),\Delta(b+d)]$$

To design the corresponding circuit for interval addition, let $X=[x_l,x_u]$, $Y=[y_l,y_u]$ be two intervals. The implementation is simply consists of two adders to produce the sum of the lower and upper endpoints of the two intervals. The sum is given by,

$$Z=[z_l,z_u]=X+Y=[\nabla(x_l+y_l),\Delta(x_u+y_u)]$$

The hardware implementation for the interval adder can be shown by the following circuit (Figure 4-1):

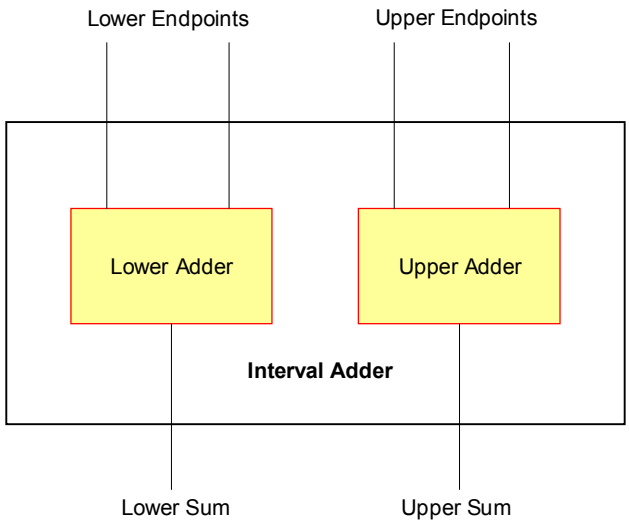


Figure 4-1: Hardware Implementation for the Interval Adder

4.2. Interval Squaring Circuit:

Let $X = [x_l, x_u]$ be an interval number, the square of X denoted X^2 is defined by:

$$X^2 = [\min(x_l^2, x_u^2), \max(x_l^2, x_u^2)]$$

The hardware implementation for X^2 can be shown by the following circuit (Figure 4-2):

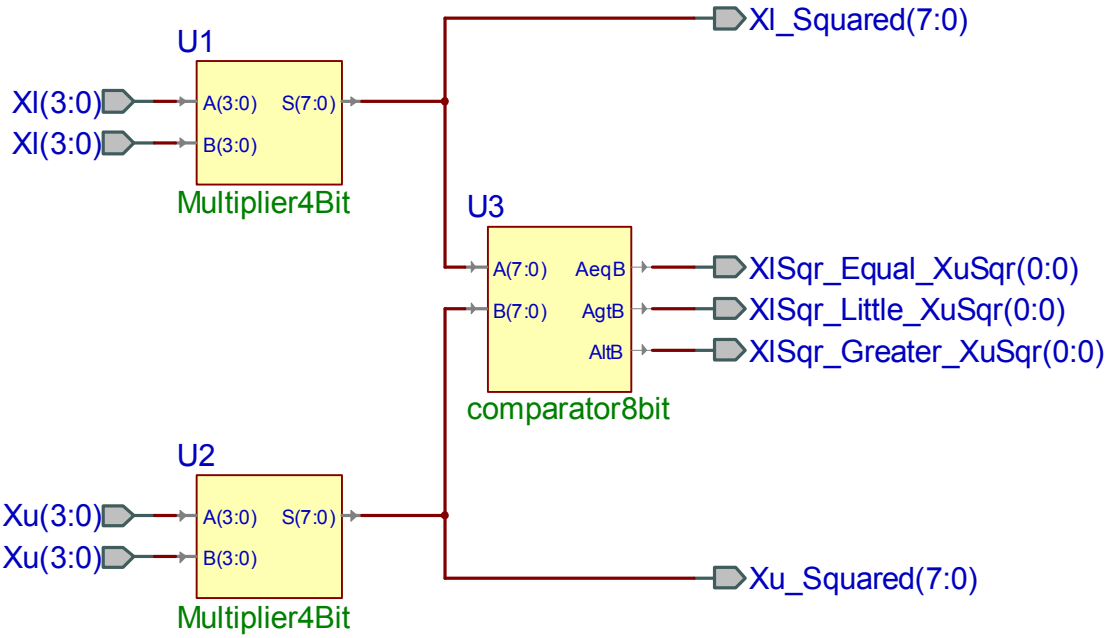


Figure 4-2: Hardware Implementation for the Interval Squaring Circuit

From this implementation we see that if x_l, x_u are two n-bit numbers, we need two $n \times n$ multipliers and one $2n \times 2n$ comparator. As an example, if x_l, x_u are two 4-bit numbers, we need two 4-by-4 bit multipliers and one 8-by-8 bit comparator. In what follows we discuss the design of the 4-by-4 bit multiplier in some detail.

The 4-by-4Bit Multiplier:

We can construct a combinational circuit for implementing the 4-by-4 bit multiplier using a method called *partial product accumulation*. Let the two numbers involved in a multiplication are called the *multiplicand* and the *multiplier*.

Let the multiplicand bits be A_0, A_1, A_2, A_3 and the multiplier bits B_0, B_1, B_2, B_3, then the multiplication of A and B becomes:

$$
\begin{array}{ccccccc}
 & & & A_3 & A_2 & A_1 & A_0 \\
 & & & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & A_3 \bullet B_0 & A_2 \bullet B_0 & A_1 \bullet B_0 & A_0 \bullet B_0 \\
 & & A_3 \bullet B_1 & A_2 \bullet B_1 & A_1 \bullet B_1 & A_0 \bullet B_1 & \\
 & A_3 \bullet B_2 & A_2 \bullet B_2 & A_1 \bullet B_2 & A_0 \bullet B_2 & & \\
A_3 \bullet B_3 & A_2 \bullet B_3 & A_1 \bullet B_3 & A_0 \bullet B_3 & & & \\
\hline
S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0
\end{array}
\qquad (4.1)
$$

Each of the ANDed terms is called a *partial product*. The resulting product is formed by accumulating down the columns of partial products, propagating the carries from the rightmost columns to the left.
The circuit for the 4-by-4 bit multiplier is shown in Figure 4-3:

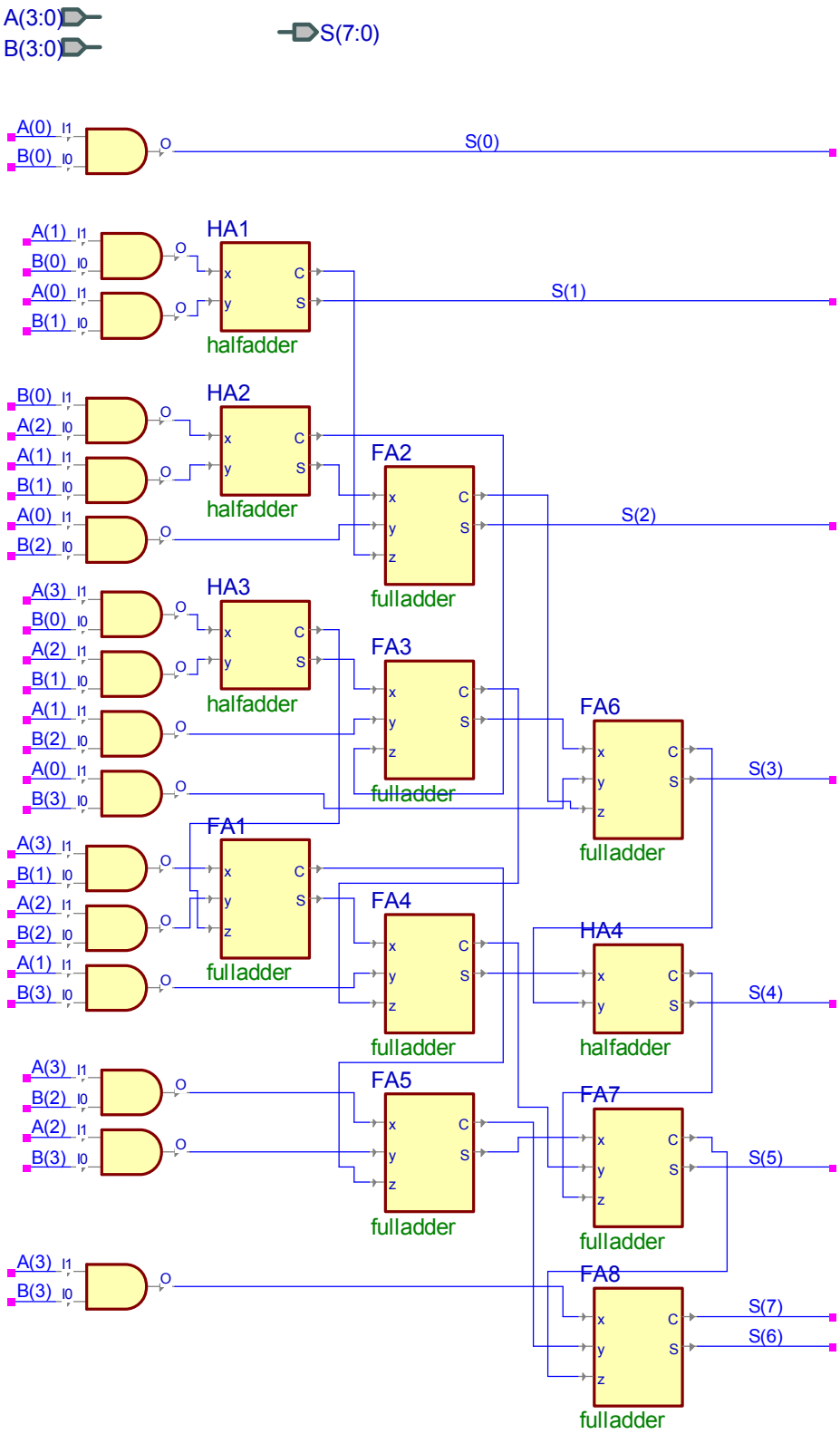


Figure 4-3: Schematic Diagram of the 4-by-4 Bit Multiplier

The first level of 16 AND gates computes the individual partial products. The second- and third-level logic blocks form the accumulation of the products on a column-by-column basis. The column sums are formed by a mixture of cascaded half adders and full adders. In the figure, inputs from the top are the bits to be added and the input from the right is the carry-in. The output from the bottom is the sum and to the left is the carry-out.

In appendices A and B, we provide a detailed description of the 4-by-4 bit multiplier and the interval squaring circuit. We introduce the schematic diagrams, the verilog descriptions of the designs, the simulation waveform views, and the verilog descriptions of the simulations.

5. More Scientific and Engineering Applications of Interval Arithmetic

More widespread real-world applications of interval arithmetic have appeared in recent years. A relatively early commentary on the use of interval methods in real world problems appeared in 1990 by G. F. Corliss in his paper "*Industrial Applications of Interval Techniques*". Since then, use of interval methods has blossomed. Interval arithmetic algorithms have been successfully applied to several applications such as quality control, global optimization, economics, quantum mechanics, artificial intelligence, and chaotic systems.
The connection between computing and mathematics provided by intervals makes it possible to solve nonlinear problems that cannot be efficiently solved using traditional floating point arithmetic. Brief selections of some scientific and engineering applications of interval arithmetic are outlined here [28] [29] [30] [31] [32]:

Electrical Engineering: Interval methods, besides providing validated results, are hundreds of times faster than a Monte Carlo method for solving AC network equations. Also, interval computations are applied in quality control in the manufacture of radioelectric devices.

Control Theory: Interval linear algebra is used to analyze Hurwitz stability, etc. in control theory applications.

Remote Sensing and GISs: Interval methods are used to take account of bounded errors in the data in decisions based on remote sensing. Also, interval methods are used in sensitivity analysis in geographic information systems (GISs).

Quality Control: Interval methods are used for quality control in manufacturing processes in which the factors fluctuate within bounds.

Economics: Linear interval methods are used in modeling economic systems to determine the effects of uncertainties in input parameters, and to include the effects of forecast uncertainties.

Artificial Intelligence and Expert Systems: Interval-valued inference was developed to handle different logical properties of knowledge representations in many expert systems.

Dynamical and Chaotic Systems: Interval techniques were used to verify that computed numerical solutions to chaotic dynamical systems are close to actual solutions with initial conditions that are near the initial conditions of the numerical solution. Also, cell-mapping methods based on classical interval arithmetic were used to robustly visualize strange attractors (SAs) in discrete chaotic systems.

Computer Graphics and Computational Geometry: Interval algorithms have been developed to handle many problems in computer graphics and computational geometry. Operations on geometric objects such as rendering, surface intersection, and hidden line removal require robustness in nonlinear equation solvers that can be provided by interval computations. A set of tools and techniques based on interval arithmetic and affine geometry has been developed to improve robustness in such operations.

6. Epilogue: Advantages and Disadvantages of Interval Arithmetic

This report has briefly introduced the subject of interval arithmetic. We introduced the theoretical aspects of interval arithmetic and a number of its applications for digital computing, as well as some widespread applications for other scientific and engineering fields. Also, we provided a discussion of some implementations of interval arithmetic at hardware level.
Despite the accuracy and self-validation of interval arithmetic, it has some disadvantages that have to be taken into account. In what follows, we list some of the advantages and disadvantages of interval arithmetic [33].

Advantages of Interval Arithmetic:

- With interval arithmetic, it is possible to automatically perform rigorous error analysis by computing mathematical bounds on the set of all possible problem solutions, and solve nonlinear problems that were previously thought to be impossible to solve.
- By using interval algorithms to solve nonlinear problems, more accurate mathematical models of physical phenomena become practical.
- Interval arithmetic is arguably the best and most efficient way to safely translate ever-increasing computer speed into mission-critical problem solutions that are otherwise impractical or impossible to obtain.
- Important interval algorithms are naturally parallel, because they progress by deleting regions where solutions are proved not to exist. Intervals provide the only known general algorithms that achieve linear speedup as the number of processors increases in parallel computing systems.

Disadvantages of Interval Arithmetic:

- Interval arithmetic requires a much higher level of abstraction than languages like Fortran-77, Pascal or C can provide.
- The main reason for low acceptance of interval arithmetic is the lack of proper hardware support that makes the arithmetic process slow. While conventional floating-point arithmetic nowadays is provided by fast hardware, interval arithmetic has to be simulated by software routines based on interval arithmetic.
- Another severe shortcoming, which makes interval arithmetic slow, is the fact that the standardized committee has accepted no reasonable interface to the programming languages so far. Nevertheless, they are not generally taken into account when the speed of interval methods is judged. Interval methods are not slow per se. It is the actual available arithmetic on the existing processors, which make them slow.

Appendix A

Design and Simulation of the 4-by-4 Bit Multiplier

In this appendix, we introduce how to use 4 half adders and 8 full adders to design a 4-by-4 bit multiplier. We introduce the schematic diagram, the verilog description of the design, the simulation waveform view, and the verilog description of the simulation.

The designs and simulations introduced in this report were implemented using the Active-HDL 7.1 integrated environment. Figure A-1 shows the main screen and the design browser window of our workspace.

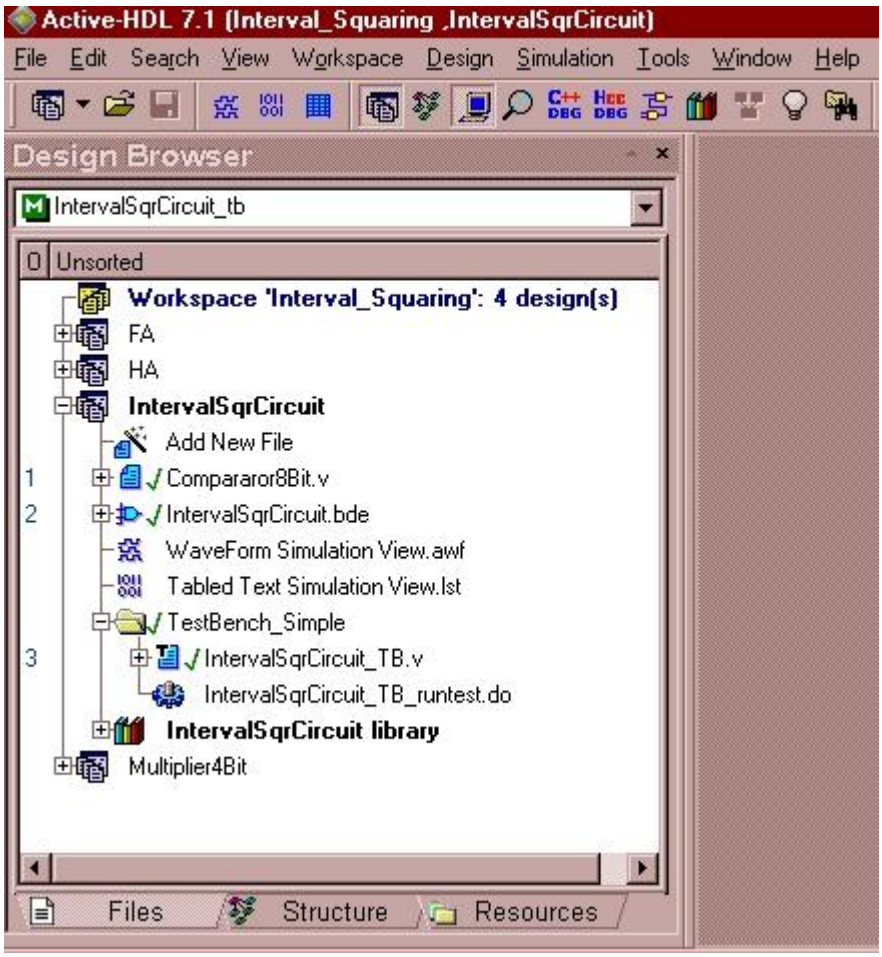


Figure A-1: The Active-HDL 7.1 IDE

A.1. Schematic Diagram of the 4-by-4 Bit Multiplier:

The implementation of the 4-by-4 bit multiplier consists of three levels of full and half adders:

- Level 1: 3 half adders, and 1 full adder.
- Level 2: 4 full adders.
- Level 3: 3 full adders, and 1 half adder.

The schematic diagram of the 4-by-4 bit Multiplier is shown in Figure A-2.

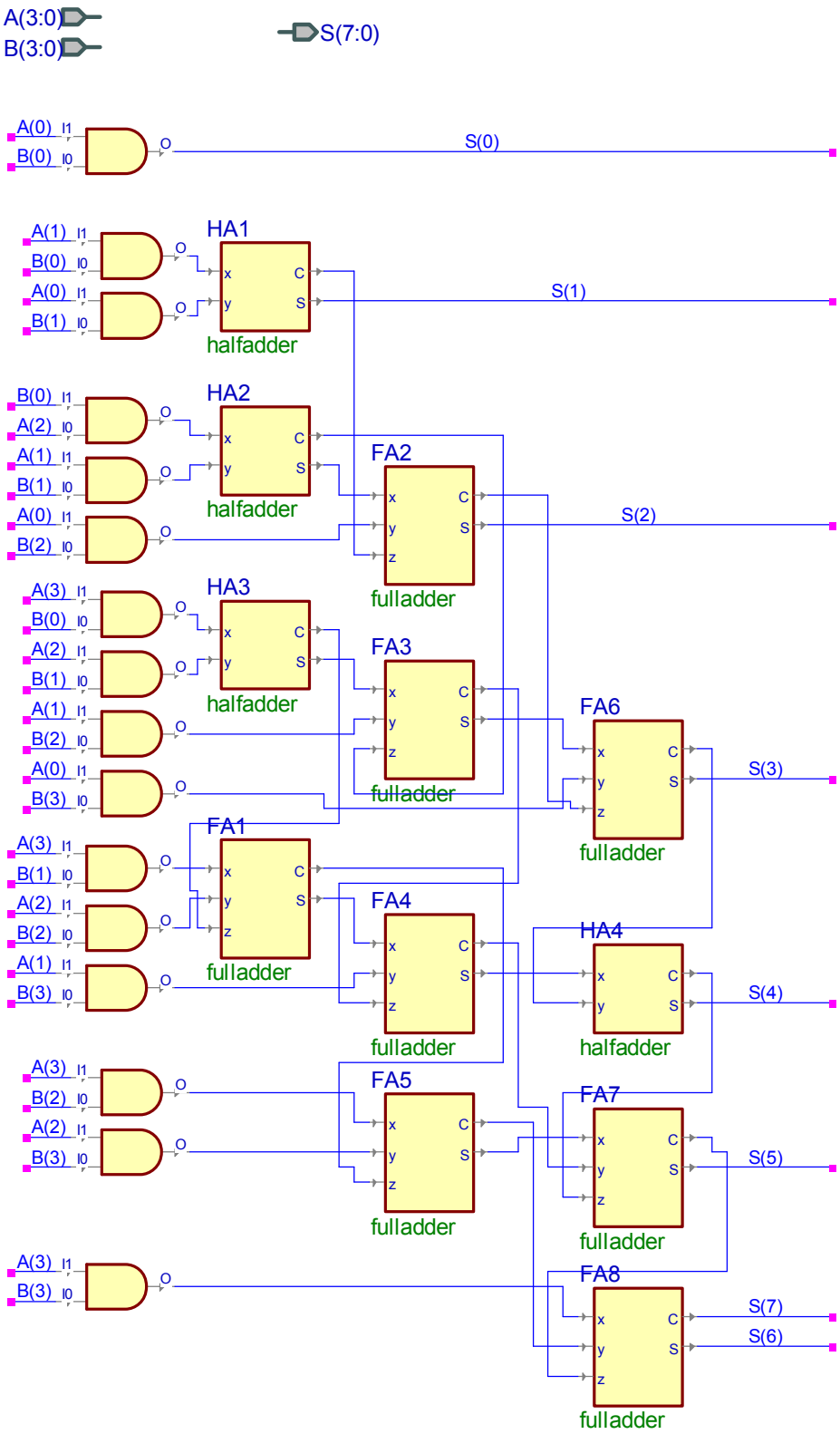


Figure A-2: Schematic Diagram of the 4-by-4 Bit Multiplier

A.2. Verilog Description of the 4-by-4 Bit Multiplier:

In this section, we list the verilog description of the half adder, the full adder, and the 4-by-4 bit multiplier.

Verilog Description of the Half Adder:

```
//------- The Half Adder ------------//
// ---- Author: Hend Dawood ---- //
//-------------------------------------//

 // ----- A halfadder module.
 module halfadder (S, C, x, y);
    input    x, y;
    output   S,C;
 // Instantiate Primitive Gates
    xor (S,x,y);
    and (C,x,y);
 endmodule
```

Verilog Description of the Full Adder:

```
//------- The Full Adder ------------//
// ---- Author: Hend Dawood ---- //
//------------------------------------ //

 // ----- A Half Adder module.
 module halfadder (S,C,x,y);
    input    x,y;
    output   S,C;
 // Instantiate Primitive Gates
    xor (S,x,y);
    and (C,x,y);
 endmodule

 //----- Description of Full Adder.
 module fulladder (S,C,x,y,z);
    input    x, y, z;
    output   S,C;
    wire S1,D1,D2; //Outputs of first XOR and two And Gates
 // Instantiate the halfadder
 halfadder HA1 (S1,D1,x,y),
         HA2(S,D2,S1,z);
 or g1(C,D2,D1);
 endmodule
```

Verilog Description of the 4-by-4 Bit Multiplier:

```
//------------------------------------------------------//
// Title      : The 4-by-4 Bit Multiplier
// Author     : Hend Dawood
//------------------------------------------------------//

`ifdef _VCP
`else
`define library
`endif


// ---------- Design Unit Header ---------- //
`timescale 1ps / 1ps

module Multiplier4Bit (A,B,S) ;

// ------------ Port declarations --------- //
input [3:0] A;
wire [3:0] A;
input [3:0] B;
wire [3:0] B;
output [7:0] S;
wire [7:0] S;

// ----------- Signal declarations -------- //
wire NET1017;
wire NET1025;
wire NET1033;
wire NET1041;
wire NET1049;
wire NET1057;
wire NET1111;
wire NET1115;
wire NET1123;
wire NET1204;
wire NET1212;
wire NET1224;
wire NET1266;
wire NET1274;
wire NET1282;
wire NET1359;
wire NET1367;
wire NET1375;
wire NET1460;
```

```
wire NET1480;
wire NET1488;
wire NET1643;
wire NET1651;
wire NET1663;
wire NET4304;
wire NET4308;
wire NET4357;
wire NET4361;
wire NET4369;
wire NET4508;
wire NET4516;
wire NET4524;

// -------- Component instantiations -------//

// synopsys translate_off
`library("FA1","FA")
// synopsys translate_on
fulladder FA1
(
    .C(NET1488),
    .S(NET1359),
    .x(NET1111),
    .y(NET1115),
    .z(NET1123)
);



// synopsys translate_off
`library("FA2","FA")
// synopsys translate_on
fulladder FA2
(
    .C(NET1663),
    .S(S[2]),
    .x(NET1204),
    .y(NET1212),
    .z(NET1224)
);



// synopsys translate_off
`library("FA3","FA")
```

```
// synopsys translate_on
fulladder FA3
(
    .C(NET1375),
    .S(NET1643),
    .x(NET1266),
    .y(NET1274),
    .z(NET1282)
);



// synopsys translate_off
`library("FA4","FA")
// synopsys translate_on
fulladder FA4
(
    .C(NET4361),
    .S(NET4304),
    .x(NET1359),
    .y(NET1367),
    .z(NET1375)
);



// synopsys translate_off
`library("FA5","FA")
// synopsys translate_on
fulladder FA5
(
    .C(NET4516),
    .S(NET4357),
    .x(NET1460),
    .y(NET1480),
    .z(NET1488)
);



// synopsys translate_off
`library("FA6","FA")
// synopsys translate_on
fulladder FA6
(
    .C(NET4308),
```

```
    .S(S[3]),
    .x(NET1643),
    .y(NET1651),
    .z(NET1663)
);



// synopsys translate_off
`library("FA7","FA")
// synopsys translate_on
fulladder FA7
(
    .C(NET4524),
    .S(S[5]),
    .x(NET4357),
    .y(NET4361),
    .z(NET4369)
);



// synopsys translate_off
`library("FA8","FA")
// synopsys translate_on
fulladder FA8
(
    .C(S[7]),
    .S(S[6]),
    .x(NET4508),
    .y(NET4516),
    .z(NET4524)
);



// synopsys translate_off
`library("HA1","HA")
// synopsys translate_on
halfadder HA1
(
    .C(NET1224),
    .S(S[1]),
    .x(NET1017),
    .y(NET1025)
);
```

```
// synopsys translate_off
`library("HA2","HA")
// synopsys translate_on
halfadder HA2
(
    .C(NET1282),
    .S(NET1204),
    .x(NET1033),
    .y(NET1041)
);



// synopsys translate_off
`library("HA3","HA")
// synopsys translate_on
halfadder HA3
(
    .C(NET1123),
    .S(NET1266),
    .x(NET1049),
    .y(NET1057)
);



// synopsys translate_off
`library("HA4","HA")
// synopsys translate_on
halfadder HA4
(
    .C(NET4369),
    .S(S[4]),
    .x(NET4304),
    .y(NET4308)
);



assign S[0] = B[0] & A[0];
assign NET1651 = B[3] & A[0];
assign NET1111 = B[1] & A[3];
assign NET1115 = B[2] & A[2];
```

```
assign NET1367 = B[3] & A[1];
assign NET1460 = B[2] & A[3];
assign NET1480 = B[3] & A[2];
assign NET4508 = B[3] & A[3];
assign NET1017 = B[0] & A[1];
assign NET1025 = B[1] & A[0];
assign NET1033 = A[2] & B[0];
assign NET1041 = B[1] & A[1];
assign NET1212 = B[2] & A[0];
assign NET1049 = B[0] & A[3];
assign NET1057 = B[1] & A[2];
assign NET1274 = B[2] & A[1];

endmodule
```

A.3. Simulation of the 4-by-4 Bit Multiplier:

In this section we describe the testing and simulation process of the 4-by-4 bit multiplier. We list here the waveform simulation view, the tabled text simulation view, and the verilog description of the simulation process. The Input-Output properties used to produce the simulation views are:

Inputs:

- A(3:0): Random Exponential Input at Period 10 ns.
- B(3:0): Random Poisson Input at Period 20 ns.

Output:

- S(7:0): Multiplication Product.

Waveform Simulation View of the 4-by-4 Bit Multiplier:

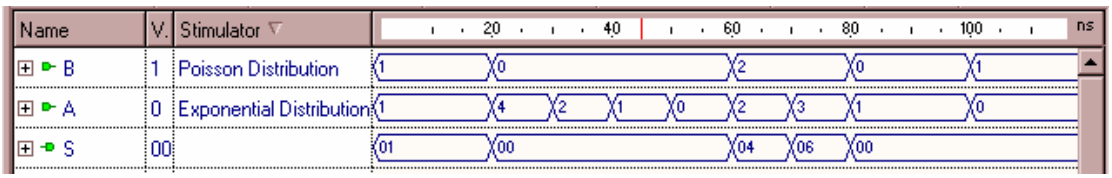


Figure A-3: Waveform Simulation View of the 4-by-4 Bit Multiplier

Tabled Text Simulation View of the 4-by-4 Bit Multiplier:

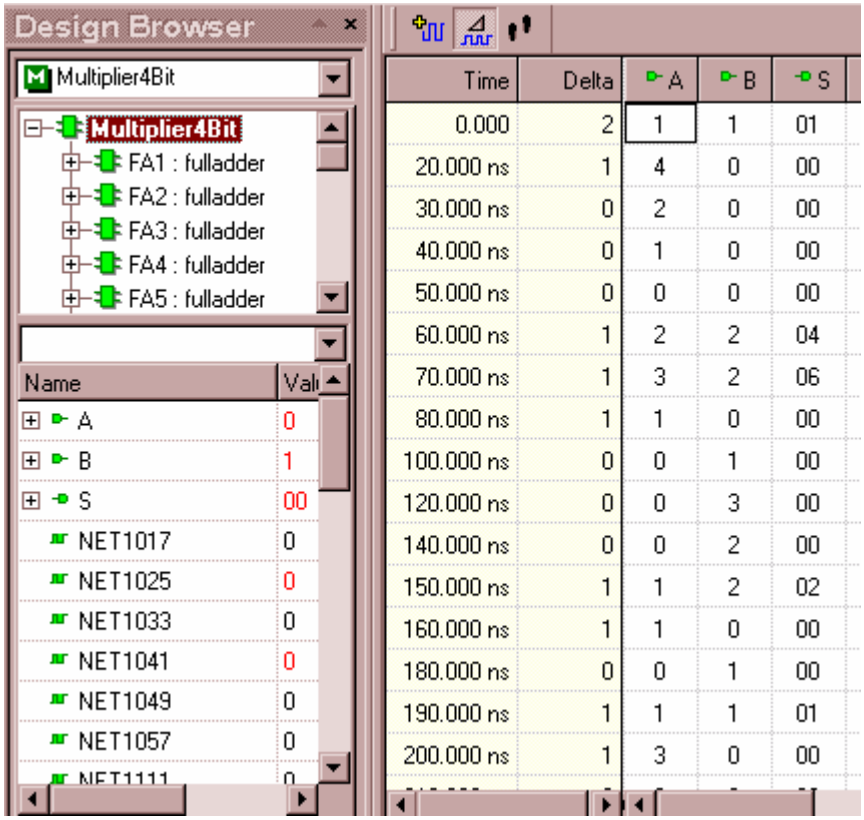


Figure A-4: Tabled Text Simulation View of the 4-by-4 Bit Multiplier

Verilog Description of the Simulation of the 4-by-4 Bit Multiplier:

```
//------------------------------------------------------//
// Title      : 4-by-4 Bit Multiplier Test Bench
// Author     : Hend Dawood
//------------------------------------------------------//

 `timescale 1ps / 1ps
 module Multiplier4Bit_tb;


 //Internal signals declarations:
 reg [3:0]B;
 reg [3:0]A;
 wire [7:0]S;
```

```
// Unit Under Test port map
    Multiplier4Bit UUT (
      .B(B),
      .A(A),
      .S(S));

initial
    $monitor($realtime,,"ps %h %h %h ",B,A,S);

//Below code was generated based on waveform file:
"f:\IAProject\Interval_Squaring\Multiplier4Bit\compile\Waveform Simulation.ver"

initial
begin : STIMUL // begin of stimulus process
    #0
    A = 4'b0001;
    B = 4'b0001;
    #20000; //0
    A = 4'b0100;
    B = 4'b0000;
    #10000; //20000
    A = 4'b0010;
    #10000; //30000
    A = 4'b0001;
    #10000; //40000
    A = 4'b0000;
    #10000; //50000
    A = 4'b0010;
    B = 4'b0010;
    #10000; //60000
    A = 4'b0011;
    #10000; //70000
    A = 4'b0001;
    B = 4'b0000;
    #20000; //80000
    A = 4'b0000;
    B = 4'b0001;
    #20000; //100000
    B = 4'b0011;
    #20000; //120000
    B = 4'b0010;
    #10000; //140000
    A = 4'b0001;
    #10000; //150000
    B = 4'b0000;
    #20000; //160000
```

```
A = 4'b0000;
B = 4'b0001;
#10000; //180000
A = 4'b0001;
#10000; //190000
A = 4'b0011;
B = 4'b0000;
#10000; //200000
A = 4'b0000;
#10000; //210000
B = 4'b0010;
#10000; //220000
A = 4'b0001;
#10000; //230000
B = 4'b0001;
#20000; //240000
A = 4'b0000;
#10000; //260000
A = 4'b0001;
#10000; //270000
A = 4'b0010;
B = 4'b0000;
#30000; //280000
A = 4'b0001;
#30000; //310000
A = 4'b0000;
B = 4'b0001;
#10000; //340000
A = 4'b0001;
#10000; //350000
B = 4'b0010;
#10000; //360000
A = 4'b0011;
#10000; //370000
A = 4'b0100;
B = 4'b0000;
#10000; //380000
A = 4'b0010;
#10000; //390000
A = 4'b0000;
#10000; //400000
A = 4'b0001;
#10000; //410000
A = 4'b0101;
#10000; //420000
A = 4'b0001;
```

```
    #30000; //430000
    B = 4'b0001;
    #20000; //460000
    B = 4'b0000;
    #10000; //480000
    A = 4'b0010;
    #10000; //490000
    A = 4'b0000;
    #20000; //500000
    B = 4'b0001;
    #10000; //520000
    A = 4'b0001;
    #10000; //530000
    A = 4'b0000;
    #10000; //540000
    A = 4'b0001;
    #10000; //550000
    A = 4'b0000;
    #20000; //560000
    B = 4'b0011;
    #20000; //580000
    A = 4'b0001;
    B = 4'b0001;
    #20000; //600000
    A = 4'b0100;
    B = 4'b0100;
    #10000; //620000
    A = 4'b0010;
    #10000; //630000
    A = 4'b0000;
    B = 4'b0001;
    #20000; //640000
    A = 4'b0010;
    B = 4'b0000;
    #10000; //660000
    A = 4'b0000;
    #10000; //670000
    A = 4'b0001;
    B = 4'b0010;
    #20000; //680000
    A = 4'b0000;
    B = 4'b0001;
end // end of stimulus process

endmodule
```

Appendix B

Design and Simulation of the Interval Squaring Circuit

In this appendix, we introduce how to use two 4-by-4 bit Multipliers and one 8-bit comparator to design an interval squaring circuit. We introduce the schematic diagram, the verilog description of the design, the simulation waveform view, and the verilog description of the simulation.

B.1. Schematic Diagram of the Interval Squaring Circuit:

The implementation of the interval squaring circuit consists of two levels:

- Level 1: two 4-by-4 bit multipliers to produce the squares of the lower and upper endpoints of an interval $X = [x_l, x_u]$.
- Level 2: one 8-bit comparator to compare the two 8-bit squares and produce three True-False binary bits for the following comparison cases:
  a. $x_l^2 = x_u^2$.
  b. $x_l^2 < x_u^2$.
  c. $x_l^2 > x_u^2$.

The schematic diagram of the interval squaring circuit is shown in Figure B-1.

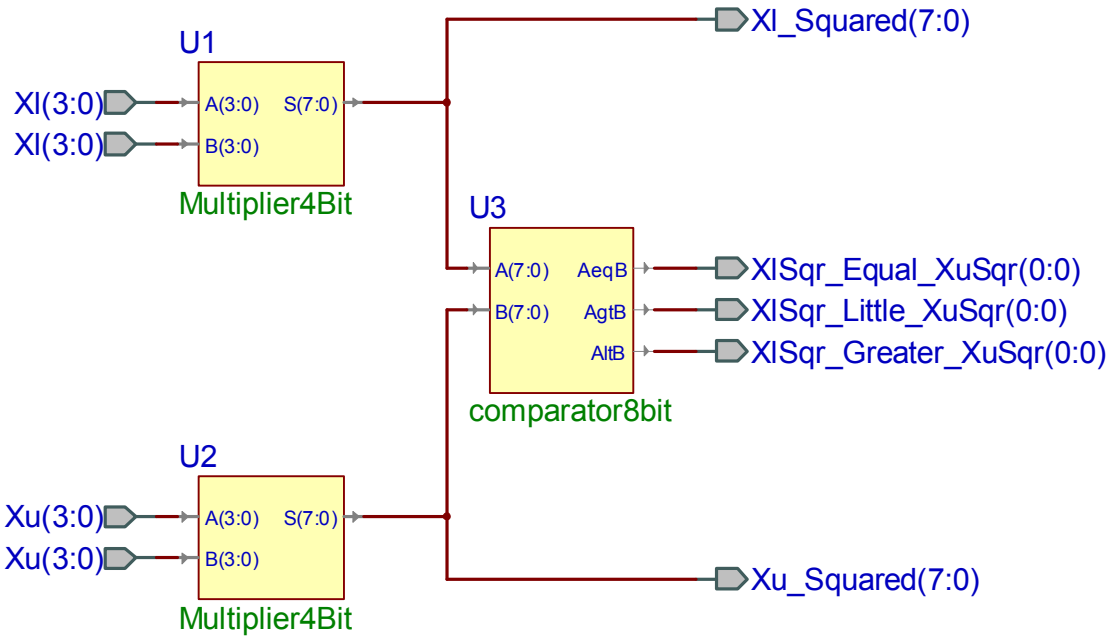


Figure B-1: Schematic Diagram of the Interval Squaring Circuit

B.2. Verilog Description of the Interval Squaring Circuit:

In this section, we list the verilog description of the 8-bit comparator, and the interval squaring circuit.

Verilog Description of the 8-Bit Comparator:

```
//------- 8-Bit Comparator ------------//
// ----- Author: Hend Dawood ------ //
//----------------------------------------//

 // ----- 8-Bit Comparator Module
 module comparator8bit (A,B,AltB,AgtB,AeqB);

    input [7:0] A,B;
    output AltB,AgtB,AeqB;

    assign AltB=(A<B),
         AgtB=(A>B),
          AeqB=(A==B);

 endmodule
```

Verilog Description of the Interval Squaring Circuit:

```
//-----------------------------------------------//
// Title      : Interval Squaring Circuit
// Author     : Hend Dawood
//-----------------------------------------------//

 `ifdef _VCP
 `else
 `define library
 `endif


 // ---------- Design Unit Header ---------- //
 `timescale 1ps / 1ps

 module IntervalSqrCircuit
(Xl,Xu,XlSqr_Equal_XuSqr,XlSqr_Greater_XuSqr,XlSqr_Little_XuSqr,Xl_Squared,Xu_Squared) ;

 // ------------ Port declarations --------- //
 input [3:0] Xl;
 wire [3:0] Xl;
 input [3:0] Xu;
 wire [3:0] Xu;
 output [0:0] XlSqr_Equal_XuSqr;
 wire [0:0] XlSqr_Equal_XuSqr;
 output [0:0] XlSqr_Greater_XuSqr;
```

```
wire [0:0] XlSqr_Greater_XuSqr;
output [0:0] XlSqr_Little_XuSqr;
wire [0:0] XlSqr_Little_XuSqr;
output [7:0] Xl_Squared;
wire [7:0] Xl_Squared;
output [7:0] Xu_Squared;
wire [7:0] Xu_Squared;

// -------- Component instantiations -------//

// synopsys translate_off
`library("U1","Multiplier4Bit")
// synopsys translate_on
Multiplier4Bit U1
(
    .A(Xl),
    .B(Xl),
    .S(Xl_Squared)
);



// synopsys translate_off
`library("U2","Multiplier4Bit")
// synopsys translate_on
Multiplier4Bit U2
(
    .A(Xu),
    .B(Xu),
    .S(Xu_Squared)
);



comparator8bit U3
(
    .A(Xl_Squared),
    .AeqB(XlSqr_Equal_XuSqr[0]),
    .AgtB(XlSqr_Little_XuSqr[0]),
    .AltB(XlSqr_Greater_XuSqr[0]),
    .B(Xu_Squared)
);


endmodule
```

B.3. Simulation of the Interval Squaring Circuit:

In this section we describe the testing and simulation process of the interval squaring circuit. We list here the waveform simulation view, the tabled text simulation view, and the verilog description of the simulation process. The Input-Output properties used to produce the simulation views are:

Inputs:

- Xl(3:0): Random Exponential Input at Period 5 ns.
- Xu(3:0): Random Poisson Input at Period 10 ns.

Output:

- Xl_Squared(7:0).
- Xu_Squared(7:0).
- XlSqr_Equal_XuSqr(0:0).
- XlSqr_Little_XuSqr(0:0).
- XlSqr_Greater_XuSqr(0:0).

Waveform Simulation View of the Interval Squaring Circuit:

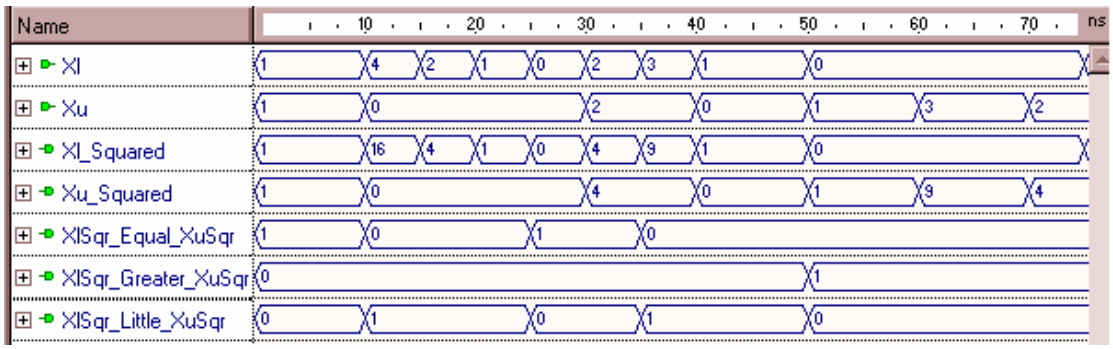


Figure B-2: Waveform Simulation View of the Interval Squaring Circuit

Tabled Text Simulation View of the Interval Squaring Circuit:

Time	Delta	Xl	Xu	Xl_Squared	Xu_Squared	XlSqr_Equal_XuSqr	XlSqr_Greater_XuSqr	XlSqr_Little_XuSqr
0.000	3	1	1	1	1	1	0	0
10.000 ns	2	4	0	16	0	0	0	1
15.000 ns	1	2	0	4	0	0	0	1
20.000 ns	1	1	0	1	0	0	0	1
25.000 ns	2	0	0	0	0	1	0	0
30.000 ns	1	2	2	4	4	1	0	0
35.000 ns	2	3	2	9	4	0	0	1
40.000 ns	1	1	0	1	0	0	0	1
50.000 ns	2	0	1	0	1	0	1	0
60.000 ns	1	0	3	0	9	0	1	0
70.000 ns	1	0	2	0	4	0	1	0
75.000 ns	1	1	2	1	4	0	1	0
80.000 ns	2	1	0	1	0	0	0	1
90.000 ns	2	0	1	0	1	0	1	0
95.000 ns	2	1	1	1	1	1	0	0

Figure B-3: Tabled Text Simulation View of the Interval Squaring Circuit

Verilog Description of the Simulation of the Interval Squaring Circuit:

```
//------------------------------------------------------------//
// Title      : Interval Squaring Circuit Test Bench
// Author     : Hend Dawood
//------------------------------------------------------------//

 `timescale 1ps / 1ps
 module IntervalSqrCircuit_tb;


 //Internal signals declarations:
 reg [3:0]Xl;
 reg [3:0]Xu;
 wire [0:0]XlSqr_Equal_XuSqr;
 wire [0:0]XlSqr_Greater_XuSqr;
 wire [0:0]XlSqr_Little_XuSqr;
 wire [7:0]Xl_Squared;
 wire [7:0]Xu_Squared;


 // Unit Under Test port map
```

```
    IntervalSqrCircuit UUT (
      .Xl(Xl),
      .Xu(Xu),
      .XlSqr_Equal_XuSqr(XlSqr_Equal_XuSqr),
      .XlSqr_Greater_XuSqr(XlSqr_Greater_XuSqr),
      .XlSqr_Little_XuSqr(XlSqr_Little_XuSqr),
      .Xl_Squared(Xl_Squared),
      .Xu_Squared(Xu_Squared));

 initial
    $monitor($realtime,,"ps %h %h %h %h %h %h %h
",Xl,Xu,XlSqr_Equal_XuSqr,XlSqr_Greater_XuSqr,XlSqr_Little_XuSqr,Xl_Squared,Xu_Squared);

 //Below code was generated based on waveform file:
"f:\IAProject\Interval_Squaring\IntervalSqrCircuit\compile\WaveForm Simulation View.vwf"

 initial
 begin : STIMUL // begin of stimulus process
    #0
    Xl = 4'b0001;
    Xu = 4'b0001;
    #10000; //0
    Xl = 4'b0100;
    Xu = 4'b0000;
    #5000; //10000
    Xl = 4'b0010;
    #5000; //15000
    Xl = 4'b0001;
    #5000; //20000
    Xl = 4'b0000;
    #5000; //25000
    Xl = 4'b0010;
    Xu = 4'b0010;
    #5000; //30000
    Xl = 4'b0011;
    #5000; //35000
    Xl = 4'b0001;
    Xu = 4'b0000;
    #10000; //40000
    Xl = 4'b0000;
    Xu = 4'b0001;
    #10000; //50000
    Xu = 4'b0011;
    #10000; //60000
    Xu = 4'b0010;
    #5000; //70000
```

```
Xl = 4'b0001;
#5000; //75000
Xu = 4'b0000;
#10000; //80000
Xl = 4'b0000;
Xu = 4'b0001;
#5000; //90000
Xl = 4'b0001;
#5000; //95000
Xl = 4'b0011;
Xu = 4'b0000;
#5000; //100000
Xl = 4'b0000;
#5000; //105000
Xu = 4'b0010;
#5000; //110000
Xl = 4'b0001;
#5000; //115000
Xu = 4'b0001;
#10000; //120000
Xl = 4'b0000;
#5000; //130000
Xl = 4'b0001;
#5000; //135000
Xl = 4'b0010;
Xu = 4'b0000;
#15000; //140000
Xl = 4'b0001;
#15000; //155000
Xl = 4'b0000;
Xu = 4'b0001;
#5000; //170000
Xl = 4'b0001;
#5000; //175000
Xu = 4'b0010;
#5000; //180000
Xl = 4'b0011;
#5000; //185000
Xl = 4'b0100;
Xu = 4'b0000;
#5000; //190000
Xl = 4'b0010;
#5000; //195000
Xl = 4'b0000;
#5000; //200000
Xl = 4'b0001;
```

```
#5000; //205000
Xl = 4'b0101;
#5000; //210000
Xl = 4'b0001;
#15000; //215000
Xu = 4'b0001;
#10000; //230000
Xu = 4'b0000;
#5000; //240000
Xl = 4'b0010;
#5000; //245000
Xl = 4'b0000;
#10000; //250000
Xu = 4'b0001;
#5000; //260000
Xl = 4'b0001;
#5000; //265000
Xl = 4'b0000;
#5000; //270000
Xl = 4'b0001;
#5000; //275000
Xl = 4'b0000;
#10000; //280000
Xu = 4'b0011;
#10000; //290000
Xl = 4'b0001;
Xu = 4'b0001;
#10000; //300000
Xl = 4'b0100;
Xu = 4'b0100;
#5000; //310000
Xl = 4'b0010;
#5000; //315000
Xl = 4'b0000;
Xu = 4'b0001;
#10000; //320000
Xl = 4'b0010;
Xu = 4'b0000;
#5000; //330000
Xl = 4'b0000;
#5000; //335000
Xl = 4'b0001;
Xu = 4'b0010;
#10000; //340000
Xl = 4'b0000;
Xu = 4'b0001;
```

```
#10000; //350000
Xu = 4'b0011;
#10000; //360000
Xu = 4'b0100;
#10000; //370000
Xu = 4'b0000;
#5000; //380000
Xl = 4'b0001;
#10000; //385000
Xl = 4'b0010;
#10000; //395000
Xl = 4'b0000;
#5000; //405000
Xl = 4'b0010;
Xu = 4'b0001;
#5000; //410000
Xl = 4'b0001;
#10000; //415000
Xl = 4'b0010;
#5000; //425000
Xl = 4'b0000;
Xu = 4'b0000;
#5000; //430000
Xl = 4'b0001;
#5000; //435000
Xu = 4'b0010;
#5000; //440000
Xl = 4'b0000;
#5000; //445000
Xl = 4'b0001;
Xu = 4'b0001;
#5000; //450000
Xl = 4'b0000;
#5000; //455000
Xl = 4'b0001;
#10000; //460000
Xl = 4'b0000;
Xu = 4'b0000;
#10000; //470000
Xl = 4'b0001;
Xu = 4'b0010;
#10000; //480000
Xl = 4'b0000;
Xu = 4'b0001;
#5000; //490000
Xl = 4'b0001;
```

```
#5000; //495000
Xl = 4'b0000;
Xu = 4'b0000;
#10000; //500000
Xl = 4'b0010;
Xu = 4'b0010;
#5000; //510000
Xl = 4'b0001;
#5000; //515000
Xl = 4'b0000;
#5000; //520000
Xl = 4'b0001;
#5000; //525000
Xu = 4'b0000;
#10000; //530000
Xl = 4'b0000;
#5000; //540000
Xl = 4'b0011;
#5000; //545000
Xl = 4'b0000;
Xu = 4'b0001;
#5000; //550000
Xl = 4'b0001;
#5000; //555000
Xl = 4'b0010;
#5000; //560000
Xl = 4'b0001;
#5000; //565000
Xl = 4'b0010;
Xu = 4'b0000;
#5000; //570000
Xl = 4'b0000;
#10000; //575000
Xl = 4'b0001;
#10000; //585000
Xl = 4'b0000;
#5000; //595000
Xl = 4'b0001;
Xu = 4'b0010;
#10000; //600000
Xu = 4'b0001;
#5000; //610000
Xl = 4'b0010;
#5000; //615000
Xl = 4'b0001;
#10000; //620000
```

```
    Xl = 4'b0000;
    #10000; //630000
    Xl = 4'b0001;
    Xu = 4'b0000;
    #10000; //640000
    Xl = 4'b0000;
    Xu = 4'b0001;
    #5000; //650000
    Xl = 4'b0001;
    #5000; //655000
    Xl = 4'b0000;
    #5000; //660000
    Xl = 4'b0001;
    #5000; //665000
    Xu = 4'b0010;
    #10000; //670000
    Xl = 4'b0000;
    Xu = 4'b0001;
    #10000; //680000
    Xl = 4'b0010;
    #5000; //690000
    Xl = 4'b0100;
    #5000; //695000
    Xl = 4'b0000;
    Xu = 4'b0010;
end // end of stimulus process


endmodule
```

References

[1] J. C. Burkill, *Functions of Intervals*, Proceedings of the London Mathematical Society, vol. 22, 1924, pp. 375-446.

[2] R. C. Young, *The Algebra of Many-Valued Quantities*, Math. Ann., vol. 104, 1931, pp. 260-290.

[3] T. Sunaga, *Theory of an Interval Algebra and its Application to Numerical Analysis*, Tokyo, 1958.

[4] R. E. Moore, *Automatic Error Analysis in Digital Computation*, Technical Report, Lockheed General Research Program, 1959, p. 43.

[5] R. E. Moore, *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. Thesis, Stanford University, 1962.

[6] Brian Hays, *A Lucid Interval*, American Scientist Magazine, Vol. 91, No. 6, December 2003, p. 487.

[7] Ibid.

[8] R. E. Moore, *Automatic Error Analysis in Digital Computation*, Technical Report, Lockheed General Research Program, 1959, p. 43.

[9] J. A. Tupper, *Graphing Equations with Generalized Interval Arithmetic*, M.Sc. Thesis, University of Toronto, 1969, p. 23.

[10] S. Shayer, *Interval Arithmetic with Some Applications for Digital Computers*, Lockheed General Research Program, 1965, p. 4.

[11] Ibid.

[12] R. E. Boche, *An Operational Interval Arithmetic*, University of Illinois National Electronics Conference, August 1963, p. 1.

[13] Ibid.

[14] S. Shayer, *Interval Arithmetic with Some Applications for Digital Computers*, Lockheed General Research Program, 1965, p. 19.

[15] M. J. Schulte, A Variable-Precision Interval Arithmetic Processor, Ph.D. Thesis, University of Texas, 1996, p. 11.

[16] S. Shayer, *Interval Arithmetic with Some Applications for Digital*

Computers, Lockheed General Research Program, 1965, pp. 5-20.

[17] R. E. Moore, *Automatic Error Analysis in Digital Computation*, Technical Report, Lockheed General Research Program, 1959, pp. 43-50.

[18] R. E. Moore, *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. Thesis, Stanford University, 1962, pp. 3-15.

[19] S. Shayer, *Interval Arithmetic with Some Applications for Digital Computers*, Lockheed General Research Program, 1965, pp. 22-23.

[20] R. E. Boche, *Complex Interval Arithmetic with some Applications*, Lockheed General Research Program, 1966, p.11.

[21] R. E. Boche, *Complex Interval Arithmetic with some Applications*, Lockheed General Research Program, 1966, pp.13-15.

[22] R. B. Kearfott, *Interval Computations: Introduction, Uses, and Resources*, University of Southwestern Louisiana, p.4.

[23] S. Markov, *On an Interval Arithmetic and Its Applications*, Late Paper, 5th IEEE Symposium on Computer Arithmetic, University of Michigan, May 1981, pp. 275-276.

[24] Ibid, p. 277.

[25] S. Shayer, *Interval Arithmetic with Some Applications for Digital Computers*, Lockheed General Research Program, 1965, p. 28.

[26] R. E. Boche, *Complex Interval Arithmetic with some Applications*, Lockheed General Research Program, 1966, pp.20-21.

[27] R. Shettar, *Hardware Implementation of Vectorized Adaptive Interval Algorithm*, IIT Bombay, June 2004, p. 2.

[28] R. Shettar, *Hardware Implementation of Vectorized Adaptive Interval Algorithm*, IIT Bombay, June 2004, p. 10.

[29] R. B. Kearfott, *Interval Computations: Introduction, Uses, and Resources*, University of Southwestern Louisiana, pp. 9-10.

[30] A. Paiva, *Robust Visualization of Strange Attractors using Affine Arithmetic*, Departamento de Matematica, PUC-Rio, Rio de Janeiro, Brazil, January 2006, p. 1.

[31] J. A. Tupper, *Graphing Equations with Generalized Interval Arithmetic*,

M.Sc. Thesis, University of Toronto, 1969, pp. 103-107.

[32] J. Chen, *Computational Geometry: Methods and Applications*, Texas A&M University, February 1996, pp. 1-3.

[33] R. Shettar, *Hardware Implementation of Vectorized Adaptive Interval Algorithm*, IIT Bombay, June 2004, pp. 14-16.

Cairo University

- Faculty of Science.
- Department of Mathematics.
- Computer Science Division.

Graduate Report in Computer Arithmetic

Hend Dawood, July 2007