BINARY FLOATING POINT ARITHMETIC VERIFICATION USING A STANDARD LANGUAGE TO SOLVE CONSTRAINTS

By

Khaled Mohamed Abdel Maksoud Nouh

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

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

BINARY FLOATING POINT ARITHMETIC VERIFICATION USING A STANDARD LANGUAGE TO SOLVE CONSTRAINTS

By
Khaled Mohamed Abdel Maksoud Nouh

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

Under the Supervision of

Prof. Dr. Hossam A. H. Fahmy

Electronics and Communication
Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

BINARY FLOATING POINT ARITHMETIC VERIFICATION USING A STANDARD LANGUAGE TO SOLVE CONSTRAINTS

By
Khaled Mohamed Abdel Maksoud Nouh

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

Approved by the
Examining Committee

_____
Prof. Dr. Hossam A. H. Fahmy, Thesis Main Advisor

_____
Prof. Dr. Ibrahim Mohamed Qamar, Internal Examiner

_____
Prof. Dr. Ashraf M. Salem, External Examiner, Faculty of Engineering, Ain Shams University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

Engineer's Name: Khaled Mohamed AbdelMaksoud Nouh
Date of Birth: 20/7/1987
Nationality: Egyptian
E-mail: Khaled_nouh@mentor.com
Phone: 01271110627
Address: 136, El Narges 4, New cairo

Registration Date: 01/10/2010
Awarding Date: 2016
Degree: Master of Science
Department: ELECTRONICS AND COMMUNICATION ENGINEERING

Supervisors:

Prof. Dr. Hossam A. H. Fahmy

Examiners:

Prof. Dr. Hossam A. H. Fahmy
Prof. Dr. Ibrahim Mohamed Qamar
Prof. Dr. Ashraf M. Salem, Faculty of Engineering, Ain Shams University

Title of Thesis:

Binary Floating Point Arithmetic Verification using a Standard Language to Solve Constraints

Key Words:
Floating point arithmetic; Constrained Simulation

Summary:
Verification of Floating Point (FP) units is a difficult task to achieve, and the cost of post-production bugs is severe. This is due to dealing with a large bit stream of inputs; simulation based verification fails to cover all possible input combinations and hence does not guarantee a 100% bug free design. On the other hand, formal methods are efficient in verification of FP arithmetic, yet they require creating a formal model, they cannot work on an optimized version of a design and may fail with complex designs due to state space explosion.
Our framework provides a new verification methodology that uses a constraint based random technique to generate test vectors for validating binary FP arithmetic instructions. The constraints used in our verification are written in System Verilog (SV) language and can be solved with any SV constraint solver tool. For every arithmetic operation, the written constraints couple the operands, intermediate results, rounding direction and the result evaluation to comply with the FP IEEE Standard (IEEE Std 754-2008).
The new proposal is generic and can be used to verify any software or hardware binary FP design/library. Also, it proves feasibility and usefulness in finding bugs for various binary FP Arithmetic operations for single and double precision formats.

Acknowledgments

First of all, I would like to thank God for giving me the strength and perseverance to complete this research. His greatest gifts are for sure the ladies that light my life, my mother who always pushed me beyond my limits, my wife who held my hands through this tough journey, my late grandmother who guided me with her heavenly blessings and my little girl who represents the future with all what it holds ahead.

Perhaps, having an idol to look up to is one of the biggest motives to excel in work. I was blessed with two: my grandfather, this great man who started from the scratch and reached greatness by means of hard work and self-confidence, he always used to say the "great ones are those with the great deeds", and my father the most knowledgeable person who I always look up to and seek advice from.

I would like also to express my gratitude to my advisor Prof. Hossam Aly Fahmy for his continuous support, guidance and patience. I could not have imagined having a better advisor and mentor.

Last but not the least, I would like to thank my elder brother, my friends and my second family at work, specially my partner in the journey Ahmed Ismail.

Table of Contents

List of Tables

List of Figures

Abstract

Verification of Floating Point (FP) units is a difficult task to achieve, and the cost of post-production bugs is severe. This is due to dealing with a large bit stream of inputs; simulation based verification fails to cover all possible input combinations and hence does not guarantee a 100% bug free design. On the other hand, formal methods are efficient in verification of FP arithmetic, yet they require creating a formal model, they cannot work on an optimized version of a design and may fail with complex designs due to state space explosion.

Our framework provides a new verification methodology that uses a constraint based random technique to generate test vectors for validating binary FP arithmetic instructions. The constraints used in our verification are written in System Verilog (SV) language and can be solved with any SV constraint solver tool. For every arithmetic operation, the written constraints couple the operands, intermediate results, rounding direction and the result evaluation to comply with the FP IEEE Standard (IEEE Std 754-2008).

The new proposal is generic and can be used to verify any software or hardware binary FP design/library. Also, it proves feasibility and usefulness in finding bugs for various binary FP Arithmetic operations for single and double precision formats.

Chapter 1 : Introduction

1.1. Floating Point Numbers

Floating point format is a way of representing real numbers with a string of digits. It maps the infinite range of real number by a finite subset with limited precision. A floating point number can be characterized by the following:

- Sign: the polarity of the number, either positive (+), or negative (-),
- Radix: the base number for scaling, usually two (binary), ten (decimal) or sixteen (hexadecimal),
- Exponent range: the interval of the maximum and minimum power of the radix,
- Significand: also called Precision or Mantissa, it is a fixed number of significant digits in base format, a string of 4 digits "10.11" is an example of a significand of binary radix, "1.250" is an example of a significand of decimal radix, and "FFF.F" is an example of a significand in hexadecimal radix. The character '.' separates the integer part of the number from the fraction part, for binary radix example, Table (1.1) explains the evaluation of the significand "10.11":

Table 1.1 Binary Radix Significand

Digit location (i)	1	0	-1	-2
Digit weight ($radix^i = 2^i$)	2	1	0.5	0.25
Digit Value s_i	1	0	1	1
Weight ×Value = $s_i \times 2^i$	2	0	0.5	0.25
Sum = $\sum s_i \times 2^i$	2.75			

In general any floating point number is represented with the following equation:

$$(-\mathbf{1})^{sign} \times \boldsymbol{significand} \times \boldsymbol{radix}^{exponent}$$

For a Binary radix, with four digit significand, a representation of "2.75" is:

$$\mathbf{2.75|10} = \underbrace{\mathbf{1.011}}_{significand} \times \underbrace{\mathbf{2}}_{radix} \overbrace{\mathbf{1}}^{exponent}$$

Representation of a number in a floating point format depends on the radix; some numbers can be represented with finite exact precision with one choice of a radix, but are approximated with other choices, for example, ‘1.1’ is easily represented with two digit decimal precision, while converting it to binary it will give ‘1. 00011001100110011001101’ which is inexact and approximated in 24 binary precision.

Integer representation of a number has a uniform step between any two consecutive numbers, while a floating point representation can have non uniform hops. On the other hand, the range and precision of a floating point representation is much bigger than integer.

1.2. IEEE-754 representation of floating point numbers

IEEE-754 [1] is a standard that specifies formats and operations for floating-point operations. It provides a method for computing any operation with floating-point numbers which should return the same result regardless of how it is implemented and whether it is in software or hardware. Also, faults and errors, in the arithmetical processing are restricted to be reported in a consistent manner as well.

The standard specifies the different formats for binary and decimal floating-point numbers; specifies the following arithmetic operations: addition, subtraction, multiplication, division, square root and fused multiply add; includes how to convert between integer and floating-point formats, between different floating-point formats, and between floating-point formats and external representations as character sequences. Also, the standard explains different floating point exceptions and how to handle them.

1.2.1. Floating point formats

The floating point standard defines floating-point formats, which represent a finite subset of real numbers as mentioned before. Formats are characterized by their radix, precision, and exponent range. These formats represent floating-point operands or results for the operations.

The standard specifies formats for both binary and decimal representations, also exchanging between these formats are defined as well. There are three formats defined for binary and two formats defined for decimal:

- Binary in single precision (32 bits encoding), double precision (64 bits encoding) and Quadruple precision (128 bits encoding),

- Decimal in double precision (64 bits encoding), Quadruple precision (128 bits encoding).

Table (1.2) summarize different format encoding for binary and decimal supported formats:

Table 1.2 Binary and Decimal formats

	Binary format (radix=2)			Decimal format (radix=10)	
Type	Single	Double	Quadruple	Double	Quadruple
Precision	24 digits	53 digits	113 digits	16 digits	34 digits
emax	+127	+1023	+16383	+384	+6144

Also, the IEEE standard specifies for every format the maximum exponent ($emax$) and minimum exponent ($emin$), where $emin = 1 - emax$ for all formats.

The representations of floating-point data in a format consist of:

- Any number between – infinity to + infinity, its value is evaluated by sign, exponent significand as explained in section 1.1 with the following equation:

$$(-\mathbf{1})^{\boldsymbol{sign}} \times \boldsymbol{significand} \times \boldsymbol{radix}^{\boldsymbol{exponent}}$$

The sign is 1 for negative numbers and 0 for positive numbers, the radix is 2 for binary and 10 for decimal, the exponent (e) is any integer such that $emin \leq e \leq emax$, the significand is a p string of digits represented as $d_0.d_{-1}d_{-2} \dots d_{-p+1}$, where d_i is an integer such that $0 \leq d_i < radix$.

- Infinities: $\pm\infty$,
- qNan (quiet), sNaN (signaling), will be explained later.

Numbers that are not infinities or NaNs are divided into two categories:

- Normal numbers: these are numbers ranging from the smallest positive normal floating-point number which is equal to $radix^{emin}$, where it is supposed that the digit left to the floating point is 1, and the largest value is $radix^{emax} \times (radix - radix^{p})$.
- Subnormal numbers: The non-zero floating-point numbers for a format with magnitude less than $radix^{emin}$, i.e. the digit left to the floating point is 0 and the exponent is $emin$. These numbers are called subnormal and their magnitudes lie between zero and the smallest normal magnitude which

is $radix^{emin}$. They always have less than p significant digits. The smallest subnormal magnitude is $radix^{emin} \times radix^{1-p}$.

Zeroes in floating point formats have an extra information stored in the sign bit. All formats have distinct representations for +0 and −0, the sign of a zero is important in some operations, for example divide by +0 leads to +infinity, while divide by -0 results in –infinity. Binary formats have only one representation each for +0 and −0, but decimal formats have more than one.

1.2.1.1. Binary floating point format

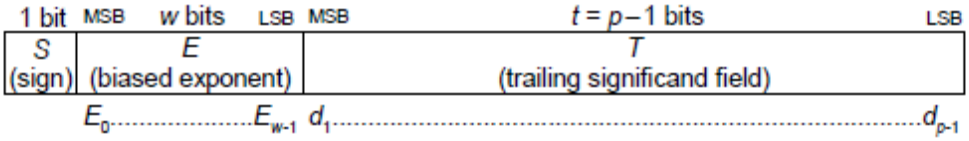

Figure 1.1 Binary Encoding Format

Figure (1.1) shows the how the binary interchange floating point format is represented. Table (1.3) summarize the values of w, E, t and T, where

- w: number of bits to represent the exponent,
- E: is the biased exponent, i.e. $E = e + bias$, and $bias = emax$, depending on its value, one can gain information about the encoded number:
  - From 1 to $2^w - 2$ → normal number,
  - 0 → ±0 or subnormal number,
  - $2^w - 1$→ encodes ±infinities and NaNs depending on the value of T
- t: is the number of bits to represent the significand, $t = p - 1$, no need to encode the left most significant digit, as for normal numbers it is always 1, while for subnormal numbers it is always 0. This un-encoded bit is usually called the hidden bit,
- T: is the value of the trailing significand, for normal numbers, $T = significand - 1 = d_{-1}d_{-2} \dots d_{-p+1}$.

Table 1.3 Different numbers in binary format

Type	E	T	Notes
Normal	$1 \leq E \leq 2^w - 2$	Any value	
Zero	$E = 0$	$T = 0$	Sign determine ±0
Subnormal	$E = 0$	$T \neq 0$	
Infinity	$E = 2^w - 1$	$T = 0$	Sign determine ±∞
NaN	$E = 2^w - 1$	$T \neq 0$	

1.2.2. Floating point arithmetic

The IEEE-754 standard supports the following arithmetic operations:

- Addition: computes $x + y$,
- Subtraction: computes $x - y$,
- Multiplication: computes $x \times y$,
- Division: computes the quotient of $\frac{x}{y}$,
- Square root: computes $\sqrt{x}$,
- Fused multiply add: computes $(x \times y) + z$,

  In fused multiply add operation, the intermediate result of multiplication is computed as if it has unbounded range and precision, rounding is done only once to the destination format after the addition operation. No underflow, overflow, or inexact exceptions are raised due to the multiplication, but only after addition; and so Fused multiply add operation differs from a multiplication that is followed by an addition,
- Convert to and from integer: The standard supports multiple operations to convert from/to integer depending on rounding technique deployed.

1.2.2.1. Arithmetic operations with operands as infinities and NaNs

Dealing with infinites is inherited from the behavior of real numbers with infinities as operands for example divide by infinity will return zero.

No exceptions arise when the operands are infinities, the following operations are valid and produce no exceptions for finite values of x:

- $x + \infty, \infty \pm x,$ returns $+\infty$
- $x - \infty$, returns $-\infty$
- $x \times \infty, \infty \times x$, returns $+\infty$, if $x \neq 0$

- $\frac{\infty}{x}$, returns ∞,
- $\frac{x}{\infty}$, returns 0,
- $x\%\infty$, return x, x is normal number
- $\sqrt{\infty}$, returns ∞

On the other hand, the following operations yield in exceptions arising:

- ∞ due to reaching the maximum value, overflow flag is raised,
- $x/0$, divide by zero exception is raised,
- $x\%\infty$, x is subnormal number, underflow flag is raised.

There are two different types of NaN, signaling (sNan) and quiet (qNaN), they are supported in all floating-point arithmetic operations. Signaling NaNs can represent uninitialized variables or arithmetic-like enhancements, it is not covered in the IEEE-754 standard. Quiet NaNs afford including diagnostic information resulting from invalid or unavailable data or results. When a certain operation is invalid, the floating point result shall be qNaN, and an invalid exception should be raises. Operations involving one or more operands as qNaN shall raise no exceptions except for Fused multiply add operation that might signal the invalid operation exception (see section 1.2.4). For most operations other than maximum and minimum with operands as qNaN inputs, the result shall be a qNaN which should be one of the input NaNs.

1.2.3. Rounding

Usually the implementation of any arithmetic operation has the intermediate result as if unbounded, and to fit the result in finite number of bits, a rounding step takes place. Following are the possible rounding directions:

- Round tie to nearest even: choose the even value that the intermediate unbounded result lie between,
- Round towards zero, the magnitude of the rounded value is less than the intermediate unbounded value,
- Round towards positive infinity, if the intermediate result is positive, the magnitude of the rounded result is greater than the intermediate unbounded result, if the intermediate result is negative, the magnitude of the rounded result is less than the intermediate unbounded result,

- Round towards negative infinity, if the intermediate result is positive, the magnitude of the rounded result is less than the intermediate unbounded result, if the intermediate result is negative, the magnitude of the rounded result is greater than the intermediate unbounded result,

1.2.4. Exception handling

There are five types of exceptions that can arise, when these exceptions happen, there exist a defined handling for the signaled exception. A corresponding status flag shall exist in an implementation abiding by the IEEE-754 standard for each kind of exception. An arithmetic operation can result in more than one exceptions, for example, exception handling for overflow and underflow signals the inexact exception. Following is the list of exceptions and how they can happen in arithmetic operations:

- **Invalid operation:**
  - An operand as sNaN,
  - Multiplication: $0 \times \infty, \infty \times 0$,
  - Fused multiply add: $(0 \times \infty) + c, (\infty \times 0) + c$, unless c is a qNaN; if c is a qNaN then the implementer is the one to judge whether the invalid operation exception should be signaled,
  - Addition/Subtraction/Fused multiply Add: subtraction of infinities, $(+\infty) \mp (\pm\infty)$,
  - Division: $\frac{0}{0}, \frac{\infty}{\infty}$,
  - Remainder: $x\%0, \infty\%x$, x is non infinity or NaN,
  - Square Root: if the operand is less than zero,
- **Divide by zero:**

  It happens if and only if an exact infinite result is defined for an operation with finite operands. The result shall be an infinity and its polarity is determined according to the operation:
  - Division, when the divisor is zero and result's sign is exclusive or of the dividend sign and the divisor sign,
  - Logarithmic: $log_2(0)$ results in $-\infty$.
- **Overflow:**

  It happens if and only if the intermediate result before rounding is greater than the format's largest finite number, depending on the rounding technique

and the result sign, the result is either infinity or maximum finite value of this format:

- If the rounding is towards zero, the result is the maximum finite value with the same sign as the intermediate result,
- If the rounding is ties to even, the result is infinity with the same sign as the intermediate result,
- If the rounding is toward positive infinity, if the intermediate result is positive, the result is positive infinity, if the intermediate result is negative, the result in the maximum negative finite value,
- If the rounding is toward negative infinity, if the intermediate result is negative, the result is negative infinity, if the intermediate result is positive, the result in the maximum positive finite value,

- **Underflow:**

  It happens when the result is a tiny non-zero value, this shall be either:

  - After rounding: when a non-zero result computed as though the exponent range were unbounded would lie between $\pm radix^{emin}$,
  - Before rounding: when a non-zero result computed as though both the exponent range and the precision were unbounded would lie between $\pm radix^{emin}$.

  When underflow occurs, a rounded result should be delivered.

- **Inexact:**

  If the rounded result varies from the intermediate unbounded result, or overflow exception is raised or underflow exception is raised, inexact flag shall be raised.

1.3. Random Simulation

Directed simulation demands generation of huge number of test vectors in order to cover all possible input combinations and hence can guarantee full coverage in testing. Yet, this is usually inapplicable specially for designs under test involving wide inputs, for example, a single precision binary floating point unit that supports only two operations and three rounding directions requires 110,680,464,442,257,309,696 test vectors ($2^{32} \times 2^{32} \times 2 \times 3$).

Usually, simulation is used with cover goals, so instead of identifying every input combination as unique test vector, a common coverage goal is defined to group some input combinations together, one naïve way of doing so is to group the inputs based on their types: ±Normal/±Subnormal/±Zero/±Infinity/±Nans, this will reduce the variation of one operand from 2^{32} to 10. Yet, this introduces a new problem: how to smartly generate the desired test vectors? The answer is Random test generation. Random test generation is a type of functional verification to provide a random stimulus to a design under test.

1.3.1. Constraint Random Test Pattern Generation

Random test generation can be more effective by specifying constraints; instead of letting the randomizer do all the work, the user can specify some constraints to hit some desired corner cases or meet a certain coverage target more easily. Figure (1.2) summarizes the flow of constraint random simulation with coverage goal.

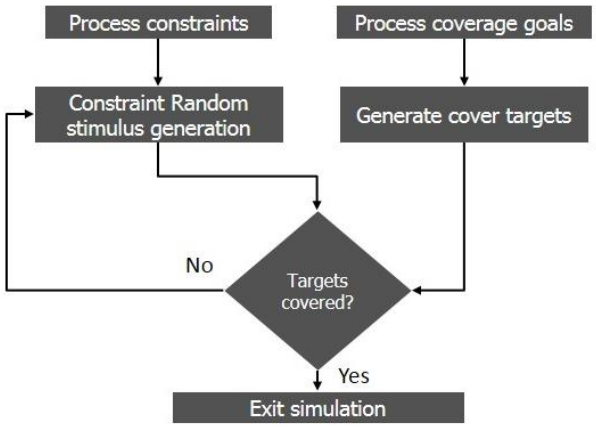

Figure 1.2 Constraint Random Simulation with Coverage Goals

1.3.2. System Verilog Constraints

Recent verification languages such as System Verilog (SV) [14] support constraint random value generation. It allows users to specify constraints in a compact, declarative way. The solver processes these constraints and consequently generates random values to meet them. The random constraints are typically specified on top of an object-oriented data abstraction (class in System Verilog Language) that models the data to be randomized as objects that contain random variables and user-defined constraints. The

constraints determine the legal values that can be assigned to the random variables. Then, a randomization function is called on the instance of the class. Table (1.4) is an example of System Verilog constraint where a class is defined for a random operation that can be an addition or subtraction and two random operands that are linked to the result based on the randomized operation.

Table 1.4 SV Constraint Example

```
1  typedef enum {add, sub} OprType;
2  class add_subtract;
3  rand OprType opr ;
4  rand int op1,op2,result;
5  constraint result_calculation {
6  (opr == add) -> (result == op1 + op2);
7  (opr == sub) -> (result == op1 - op2);
8  };
9  endclass
10 add_subtract add_sub = new ;
11 repeat (10) add_sub.randomize();
```

To explain the importance of constraints in random simulation, Table (1.5) is an SV test where we have a coverage goal that the result is 0. This will only be valid for any of the following scenarios:

1- When the operation is subtraction and the operands are equal to each other,
2- When the operation is addition and the operands are same magnitude but have opposite signs.

Table 1.5 SV Constraint Example with coverage goal

```
1  `define N 1000
2  `define LSB 5
3  module dut;
4    typedef enum {add, sub} OprType;
5    class add_subtract;
6      rand OprType opr ;
7      rand int op1,op2,result;
8      constraint result_calculation {
9        (opr == add) -> (result == op1 + op2);
10       (opr == sub) -> (result == op1 - op2);    };
11     constraint close_operands {
12       op1[31:`LSB] == op2[31:`LSB];  }
13   endclass
```

```
14   int i;
15   add_subtract a;
16    initial
17   begin
18     i  =  0;
19     a = new();
20     repeat (`N) begin
21     assert (a.randomize());
22     i++;
23     if (a.result == 0)
24       $finish;
25     end
26   end
27 endmodule
```

As seen in Table (1.5), the new constraint "close_operands" specify that part of the two operands are equal, this part is controlled by a parameter `LSB that is allowed to have values between 31 and 0. It is obvious that the smaller the `LSB is, the more close the operands will be. Table (1.6) shows the different values of `LSB and the corresponding number of test vectors required to hit the coverage goal explained earlier (denoted by lines 23 and 24 in Table (1.5).

Table 1.6 Number of tests required to hit coverage goal with respect to constraints on inputs

`LSB value	5	4	3	2	1	0
# of tests required	315	37	9	4	2	1

1.3.2.1. Random Variables in System Verilog

The IEEE-1800 standard of System Verilog supports two types of random variables:

- **rand**: Random variables with uniform distribution over their range,
- **randc**: Random variables where variables cycle through all the values in a random permutation of their declared range. For example, for the following random variable declaration:

Table 1.7 randc Random variables

```
1  randc bit [1:0] y;
2  //First permutation: 0→3→2→1
3  //Second permutation: 2→3→0→1
4  //Third permutation:2→0→1→3
```

5	// and so on…
6	//Values of y in successive calls: 0,3,2,1,2,3,0,1,2,0,1,3

The possible values of y are 0, 1, 2 and 3. Initially, the randomizer generates an initial order for y covering its whole range, and return these values in successive calls of y until the last value, a second order of y is generated and looped over until the last value of the order and so on. The main advantage of this type of randomization is that all values of y are covered before any value is repeated.

1.3.2.2. Constraint Blocks in system Verilog

System Verilog supports adding constraints to constrict the values of the declared random variables. These constraint blocks are class members and have a unique name called constraint identifier. Below is the explanation of the main features of constraint blocks:

Table 1.8 Constraint block example

```
rand integer a, b, c, x, y, z;
rand integer A[10];
rand bit s;
rand integer d;
constraint c1 {
x inside {3, 5, [8:15]};
y dist {10:=1, 20:= 2, 30:= 4};
unique {a,b,c};
(a > 10) -> (x == 3);
if (y == 10) z == a;
else if (y == 20) z == b;
else z == c;
foreach ( A [ i ] ) A[i] inside {2,4,8,16};
(s) -> (d == 0);
solve s before d;
}
```

- Setting membership

  It restrict certain values allowed to one random variable, in the example in Table (1.8), x can only have 3, 5, 8 through 15 values.

- Distribution

It sets weight to values allowed to one random variable, the possible values of y in the example in Table (1.8) are 10, 20 and 30, with the weighted ratios 1:2:4 respectively ,i.e., y is more likely to be 30 than 20, and 20 than 10.

- Uniqueness

  It restrict the embedded list to have mutually exclusive values, in the example in Table (1.8), a, b and c are not allowed to have the same value.

- Implication

  It correlates a subsequent condition given the evaluation of an antecedent condition; it the antecedent condition is true the subsequent condition should be true, if the antecedent condition is false, no restrictions are set on the subsequent condition, in the example in Table (1.8), if the value of a is greater than 10, the value of x is 3.

- If-else constraints

  It constraints set of expression given other set of expressions enclosed in if-else style, for example, the value of z is equals to a, b, c if y equals 10, 20, 30 respectively.

- Foreach iterative constraints

  Iterative constrains allow looping over elements of array variables, the foreach iterative constraints in Table (1.8) sets membership for every element of the ten elements in the array A, i.e., the possible values of any element is 2, 4, 8, 16.

- Functions in constraints

  Some constraints cannot be expressed given the above mentioned constructs, so System Verilog allows the use of function calls in constraint expressions. Table (1.9) shows how to express leading zeroes calculation for a given signifincand, it loops over the significand from the left most bit to the right most and exit with the index if the bit value is 1.

Table 1.9 Function example in SV constraint

1	**`define** p 24
2	**rand bit** [`p-1:0] Significand;
3	**rand bit** Guard, Round, Sticky;
4	**rand int** shift_left_value;
5	**function** int leading_zero_calculation (input [0:`p+2] functionSignificand);
6	**for** (int i = 0; i <= `p+2;i++) **begin**

```
7                 if (functionSignificand[i] == 1'b1) return i;
8             end
9             return `p+3;
10    endfunction
11    constraint normalize_intermediate_result {
12    shift_left_value == leading_zero_calculation({Significand,Guard,
13    Round,Sticky});
      }
```

- Variable ordering

  One can specify to the solver the order to constraint and randomize a variable, consider the constraints on s and d in the example in Table (1.8), the constraint correlates values of 2 variables: s, d, from the readers' view, it states that s implies d equal zero, from the solver's view, s and d are random variables and solving {s, d} can have $2^{32} + 1$ combinations, but s is true only in one combination. In line 16, the order of solving is specified to solve s before d, and hence s can have 2 possible combination, and then d is chosen subjected to the value of s. Adding this order constraint does not change the set of legal value combinations, but alters their probability of occurrence. Table (1.10) explains the probabilities of values of d and s.

Table 1.10 Probability of variables with and without solve order

Value of s	Value of d	Probability before solve order	Probability after solve order
1	'h00000000	$1/(1+2^{32})$	$1/2$
0	'h00000000	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$
0	'h00000001	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$
0	'h00000002	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$
0	…	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$
0	'hfffffffe	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$
0	'hffffffff	$1/(1+2^{32})$	$1/2 \times 1/2^{32}$

1.3.2.3. Randomization method

The instance of the class enclosing the constraints should be called with a randomization method in order to randomize the declared variables in this class. System Verilog defines the following randomization method:

- **Randomize ()**: The randomize () method returns 1 if it successfully sets all the random variables and objects to valid values; otherwise, it returns 0. Failure to randomize can be due to conflicting constraints.

There are other topics that are discussed in details in chapter 18 in the IEEE-1800 standard of System Verilog [14] like for example generation of random sequence, manually seeding of randomization, random weighted case, etc…. We are only concerned in previously described definitions/constructs.

1.4. Floating point verification challenges

1.4.1. Wide input

As mentioned earlier, the large bits of inputs for floating point unit makes it hard for simulation to have all possible input combinations. Hence, simulation stand alone is not a good verification environment, as it will take forever to guarantee a bug free design.

1.4.2. Usually pipelined implementation

Implementation of some floating point arithmetic operations requires huge computation and complex hardware, like iterative implementations of division and square roots that imposes delay in the logical circuits slowing down the overall frequency of the processor. This is why pipelining is introduced in floating point units.

Pipelining means that at a certain time instance, there may exist more than one execution of a FP operation sharing the same hardware. This imposes a complexity in creating the verification model for this pipelined floating point unit.

Also pipelining imposes latency to the result due to execution in sequence of number of cycles, and hence formal verification of a pipelined floating point unit may fail due to state space explosion.

Another dilemma, the cycle delays for one operation is different for another operation, which will add more complexity for verifying a floating point unit that supports multiple operations with varying number of cycles.

1.4.3. No generic solution for Software/Hardware implementation

Floating point arithmetic can exist as software libraries like a C code, can be a hardware design in the form of Register Transfer Logic (RTL) and can be an optimized

version of this hardware, like Gate Level Designs. This is a problem that stands against creating a unified way to verify floating point units in general. For example, RTL can be verified using good test benches based on UVM or by addition properties and testing the functionality using model checking formal techniques, on the other hand, software libraries are verified using programming language. So the methods used in verifying the Hardware implementation are incompatible with verifying the software version of the floating point implementation.

1.4.4. Dealing with intermediate value before rounding

Usually verification techniques are considered black-box testing, i.e. they are only interested in how to generate inputs, and how to debug the outputs. For floating point units, the result is usually a rounded version of the unbounded intermediate results, this intermediate result is the main source of all bugs in any floating point design. Therefore, verification of floating point unit should follow a white-box testing technique; it should have the capability to generate stimulus/model that can treat the intermediate unbounded result as an input to the verification environment.

1.5. Our verification framework

We propose a verification framework that combat all the above mentioned challenges, which is Test vectors generation based on a model written with system Verilog constraints. The SV model defines random variables for input, outputs, intermediate results, round directions, and constraints to constrict the data path starting from the operands through intermediate results and rounding techniques until the result evaluation abiding by the IEEE-754 standard of Binary floating point formats and operations. Then, we pass the SV constraints to a simulator to randomly generate test vectors based on the above constraint model plus adding user defined constraints to cover interesting corner cases. These test vectors can be applied to any floating point design whether it is software or hardware, pipelined or not. This method shows effectiveness in discovering bugs for addition-subtraction, multiplication, division and square root operations in different binary floating point unit designs.

1.6. Organization of the thesis

The remainder of this thesis organized as follows. 0 provides a detailed survey of the techniques used in verification of floating point arithmetic. Chapter 3 provide a detailed description of the new proposal used in verification of binary floating point arithmetic operations. Chapter 4 shows the result of our work in detailed representation and draws a comparison between other techniques. Finally, chapter 5 discusses the possible future extension of our work.

Chapter 2 : Literature Review

2.1. Introduction

Verification of Floating point units is always a challenging task. Many daily applications depend on correct floating point calculations. If a single bug is missed during development and discovered at customer site, the cost will be severe similar to the division bug in the Intel processor [17]. The first lesson learned from the Intel bug was the essentiality of more thorough testing since the bug was systematic and was not caused from random hardware issues. Yet, thorough testing is still hard to achieve due to the huge input bit stream size; simulation based verification fails to cover all possible input combinations and hence does not guarantee a 100% bug free design.

2.2. Formal verification of floating point arithmetic

Formal methods are efficient in verification of FP arithmetic, yet they require creating a formal model, they cannot work on an optimized version of a design and may fail with complex designs due to state space explosion.

Some approaches of formal verification targets verification of floating point unit in general using theorem proving, the authors in [6] propose a complete formalization of the IEEE-754 standard using higher order logic specification (an expressive formal modelling method that allows quantifying over a function of a certain variable). The formalization is done for: (1) floating point numbering format, (2) different floating point arithmetic operations namely addition, subtraction, multiplication and division, and finally (3) normalization and rounding steps of intermediate unbounded result.

Another approach is introduced in [7], where the authors extends an existing verification tool for verifying floating point arithmetic in C programs. This tool is called "Caduceus," it is a first order logic model for C programs (first order logic is an expressive formal modelling method that allows quantifying over a certain variable) [18]. The formal model for floating point arithmetic is written in Coq [19], formal proof management system. On the other hand, the authors in [27] create their own formal model based on first order logic theorem proving to formulize floating point arithmetic.

Another application that make use of the formal model in Coq, is the verification of a generic End-Around-Carry Adder that is widely used in floating point arithmetic [36].

The writers in [22] present a way to overcome the failure of theorem proving to work on rounded values, and the inability of model checking to generate a counter example when the number is in floating point format. They present an approximate way of mapping floating point numbers to integer using abstract interpretation, and verifying the integer version using traditional formal methods.

Paper [28] presents an approach based on equivalence checking which can be decomposed into two steps; the first step is to verify all the data path of the floating point unit by comparing result with a reference model, the second step is to verify pipelined floating point instruction using sequential equivalence model checking.

Some approaches use in their verification a hybrid formal verification combining model checking and theorem proving, like verification of floating point multiplier in the Intel IA-32 Pentium® microprocessor [8], and verification of a pipelined double precision Multiplier based on IEEE standard [23]. The presenters in [8] here combine two techniques of formal verification; Symbolic Model checking based on Binary Decision Diagram (BDD) - a tree based structure to model any Boolean expression- and theorem proving based on formulation of temporal logic using pre-post-condition analysis. For Multiplier, BDD stand alone is not applicable due to the exponential growing size of multiplier tree regardless of the variable order when forming the reduced order BDD [20]. Also, theorem proving standalone involves so much effort and user interaction. Similar approach is done in [21] combining word-level model checking with theorem proving to verify the hardware of Pentium Pro processor.

A new data structure is introduced in [35], it is called Multiplicative Power Hybrid Decision Diagram (*PHDD); it is a compact representation for functions that map Boolean vectors into integer and floating point values. Making use of the newly proposed data structure proved to grow linearly with the word size in multiplication, exponential in the exponent part of addition, but linear with the Mantissa. Compared to other structures such as Multiplicative Boolean Matrix Decomposition (*BMD), experimental results shows that *PHDD is 6 time faster.

Another point of concern is addressed in [24], which is involving formal verification on an optimized version of floating point circuit. The authors apply theorem proving on gate level version of the design to guarantee the correctness of the rounding step based

on Even-Seidel algorithm for addition [25] and multiplication [26] to be compliant with IEEE-754 standard.

2.3. Simulation verification of floating point arithmetic

The author in [31] presents a way to combine coverage analysis and constraint random testing to speed up the verification process by 4.5x. An automatic input generator is created at the front end of verification that feeds the design under test with test vectors that have not been covered yet. At the back end of testing, a monitor is created to capture the correctness of the results and provide a complete coverage analysis for inputs and covered corner cases.

The presenters in [33] addresses the problems of functional verification of decimal floating point adder-subtract for FPGA described in VHDL, where they develop a verification test plan and create a verification environment based on Open Verification Methodology (OVM) [32]. OVM is an open source simulation based functional verification methodology that replaces writing an application specific test benches, instead, building the testing environment using reusable verification components that are structured for use in different applications. In this context, the presenters made use of the built in checker comparison in OVM and apply it to compare the arithmetic unit output with a reference model which is decNumber C library [33].

Coverage models based on Equivalent Partition and Boundary Value analysis aim to hit corner cases and reduces the number of test vectors required to fulfill the test space [4]. The authors in [2, 3] introduce a tool called FPgen that generates random tests based on defining coverage models relating the inputs, intermediate results and the outputs.

Two behavioral tools are developed in [5]; vecgen is a C program that is used to generate floating point test vectors based on input specification file that contains description of the vectors required, and fpc is a C program that defines a model based on the IEEE standard in [1]; in order to model an implementation compliant with the IEEE-754 standard, a small amount of C code is written to make calls that will initialize fpc, execute it, and model chips issues such as pipeline delays register files, etc…. These two tools are integrated and simulated together to form an automatic verification for FP units.

Generation of floating point test vector is addressed differently in [39]; given some ranges for the operands and the result, a rounding technique, it is requested to find a

random test vector where operand and result fit in their corresponding range. This approach is completed for addition and subtraction and is partially completed for multiplication and division. The benefit of this approach, is that one can specify the range of certain operand/result to cover a certain corner case.

For iterative binary floating point operations such as division and square root, a proposal is done in [40] to randomly generate operands based on the iterative algorithm used, the iteration number and a relative error interval.

Another point of interest is how to set coverage goals on intermediate result or add constraints on it and how to generate the corresponding test vectors. Intermediate result is the result of any operation before rounding takes place assuming at this stage infinite representation of bits. The authors in [3, 10, 41] address this issue by determining the maximum number of bits in intermediate result for every arithmetic operation, then do a bit level analysis to create constraints that link intermediate bits with input/output bits, then solve those constraints. This is basically the internal algorithm of FPgen tool that the author develops to verify addition, subtraction, multiplication and division. [11, 12] use similar approach by creating their own engines to solve simultaneously constraints on unbounded intermediate results and constraints on inputs-outputs. This methodology has proven effectiveness in verifying Addition-Subtraction, Multiplication and Fused Multiply Add operations for Decimal FP units.

2.4. Hybrid techniques of floating point verification

Verification of Jaguar x86 Floating point unit was carried out using both simulation and formal verification depending on the stage of the product [29]; in RTL stage, a simulation based verification based on pseudo random test vector generation and comparing result with a reference model result, next in Core/System stage, sequence of floating point instructions was verified using an architecture level third party tool, and finally in the execution stage, formal model based on theorem proven and property model checking is used to feedback to the coverage analysis used in pseudo random simulation in the first stages and hence speed up these first stage.

Symbolic Trajectory Evaluation (STE) is used to verify Intel® Processor Graphics floating point unit [30]. STE is a formal verification model checking technique using a

symbolic simulation based approach where simulation works on the BDD version of the design and specifications.

A High Level Synthesis tool is applied to synthesize and verify a Floating point unit starting from its behavioral model until gate level is reached [37]. Simulation is applied in both the behavioral model stage and in RTL stage where in the RTL stage a third party tool is applied to generate floating point test vectors. After Gate level synthesis, another third party tool is used to create logical equivalence with RTL.

A formal model for the division operation is used in [38] to generate test vectors for floating point division. First, the algorithm generates the intermediate (unbounded) value of the quotient based on some heuristics and pseudo random generation, and work back using the formal model to generate the dividend and the divisor completing the test vector format.

2.5. Summary

Formal and simulation methods are widely used in the verification of floating point units. In our framework, we propose a verification methodology for Binary FP arithmetic operations by writing SV constraints to constrict the data path starting from the operands through intermediate results and rounding techniques until the result evaluation. Then, we pass the constraints to a simulator to randomly generate test vectors based on the above constraint model plus adding user defined constraints to cover interesting corner cases. This method shows effectiveness in discovering bugs for addition-subtraction and multiplication operations in different designs.

Chapter 3 : Proposal

Figure (3.1) illustrates the verification framework. The Operation constraints define the random variables for every operation and the constraints that link these variables together based on the arithmetic operation desired. The User constraints impose more restrictions on the defined variables to force the generation of test vectors to cover a specific scenario. Both types of constraints are simulated simultaneously to generate a number of test vectors.

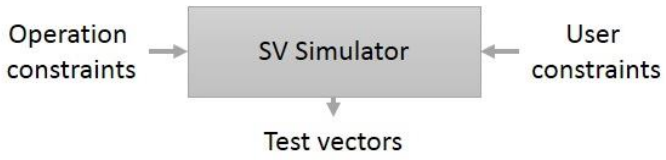

Figure 3.1 Verification Framework

In our framework, we implement the above mentioned verification framework using System Verilog Constraints and applied it on addition, subtraction, multiplication, division and fused multiply add operations. The following sections give a detailed insight about each operation and what are the aspects carried in each operation.

3.1. Addition and subtraction

Figure (3.2) summarizes the data path for the Operation constraints for Add/Subtract.

3.1.1. Encoding/decoding constraints

These constraints are concerned with mapping the binary encoding format of IEEE-754 floating point operands and result in different precisions to sign, exponent and significand. Also, it is responsible for figuring out whether the hidden bit of the significand is 1 or 0. The hidden bit is constraint with the value of the exponent, if the value of the exponent is 0, then the hidden bit is 0, else, the hidden bit is 1. Table (3.1) is the SV code for this constraint process:

Table 3.1 Encoding/Decoding constraints

```
1  rand bit [`k-1:0] operand1,operand2,result; //k is the width of the operands
2  rand bit operand1Sign,operand2Sign,resultSign;
3  rand bit [`w-1:0] operand1Exponent,operand2Exponent,resultExponent;
4  rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
5  rand bit [`p-1:0] operand1Significand,operand2Significand,resultSignificand;
6  constraint binary_encoding_decoding {
7          {operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
8          {operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
9          {resultSign,resultExponent,resultMantissa} == result;
10 }
11 constraint significand_mantissa {
12         if (operand1Exponent == ‘0)
13                 ({1'b0,operand1Mantissa} == operand1Significand);
14         else
15                 ({1'b1,operand1Mantissa} == operand1Significand);
16         if (operand1Exponent == ‘0)
17                 ({1'b0,operand2Mantissa} == operand2Significand);
18         else
19                 ({1'b1,operand2Mantissa} == operand2Significand);
20 }
```

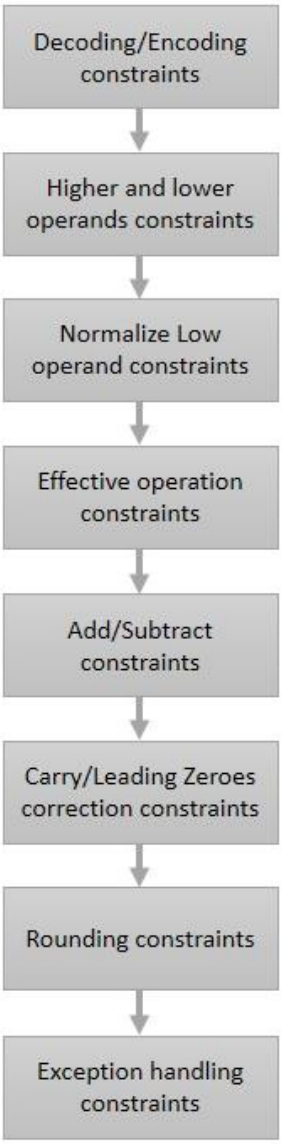

Figure 3.2 Add/Subtract operations constraints

3.1.2. Higher and lower operands constraints

These constraints are responsible for detecting which operand is the higher and which is the lower. This is important for the next constraints set in section 3.1.3. The flow chart in Figure (3.3) summarize the work flow of these constraints.

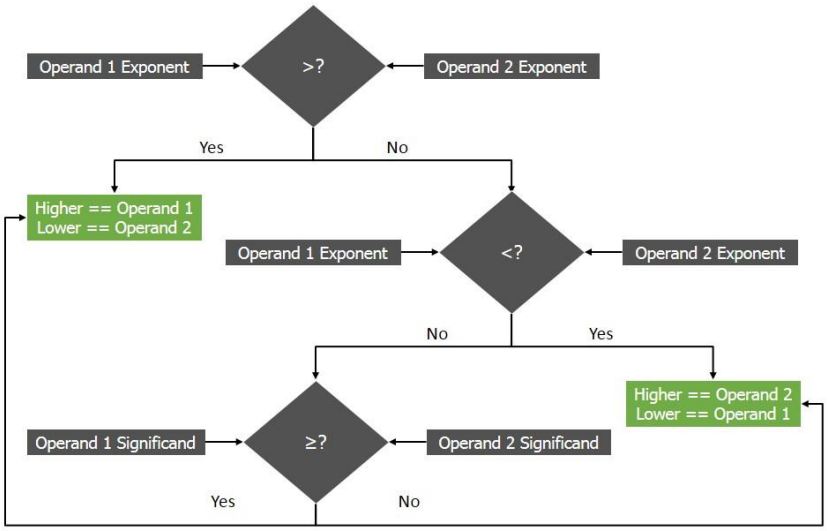

Figure 3.3 Flow for picking the higher and lower operands

3.1.3. Normalize low operand constraints

After determining the lower operand, some normalization constraints should be carried on by right shifting the operand significand until the lower exponent matches the higher exponent. Also, three extra bits are introduced here that will be used in later stages:

- **Guard bit:**

  The left most bit of the shifted out part of the lower significand,

- **Round bit:**

  The second left most bit of the shifted out part of the lower significand,

- **Sticky bit:**

  It represents all the remaining bits of the shifted out part of the lower significand, sticky bit is the disjunction of these bits.

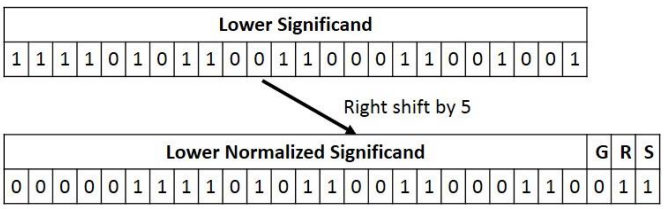

Figure 3.4 Lower significand normalization

Figure (3.4) is an example of an exponent difference between the higher and the lower operand equals to 5, the lower operand is shifted to right 5 places, 2 of which are

placed in the Guard (G) and the Round (R) bits, and the remaining 3 bits are ored together to produce the Sticky bit (S).

3.1.4. Effective operation constraints

Depending on the operation, the sign of the higher and the lower operands, the effective operation constraint is evaluated. The effective operation is addition if the operation is add and the sign of both operands is the same, or the operation is subtraction and the sign in different. Otherwise, the effective operation is subtraction.

Table 3.2 Effective operation constraints

```
1   typedef enum {ADD,SUB} operationTypes;
2   rand operationTypes operation, effectiveOperation;
3   constraint effective_operation {
4           if ((operation == ADD) && (higherSign == lowerSign))
5                   (effectiveOperation == ADD);
6           else if ((operation == ADD) && (higherSign != lowerSign))
7                   (effectiveOperation == SUB);
8           else if ((operation == SUB) && (higherSign == lowerSign))
9                   (effectiveOperation == SUB);
10          else
11                  (effectiveOperation == ADD);
12  }
```

3.1.5. Add/Subtract constraints

Based on the effective operations, the significand of the higher and lower normalized operands are added/subtracted from one another, generating the intermediate result significand. This intermediate result significand is expected to be unbounded, and hence some random variables are introduced in this steps which are:

1- Carry:

   A bit added to the left of the intermediate result significand to detect if an overflow in addition stage occurs (this doesn't mean that overflow flag is issued), this carry will be used later to normalize the intermediate result.

2- Intermediate Guard, Round and Sticky bits

   These bits will be used in rounding later.

At this point, the intermediate result exponent is constrained to be the same as the higher exponent, the intermediate result sign is determined from the effective operation and the sign of both operands as shown in Table (3.3).

Table 3.3 Intermediate result sign constraint

```
1  if ((effectiveOperation == ADD) && (operation == ADD))
2     intermediateSign == higherSign;
3  else if ((effectiveOperation == ADD) && (operation == SUB))
4      intermediateSign == operand1Sign;
5  else if ((effectiveOperation == SUB) && (operation == ADD))
6      intermediateSign == higherSign;
7  else
8     intermediateSign == ((operand1 < operand2)^(operand1Sign));
```

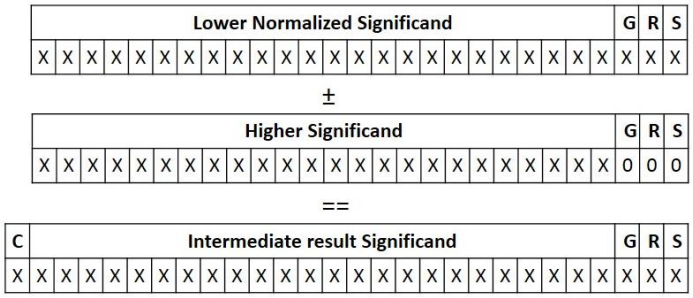

Figure 3.5 Intermediate significand constraint

3.1.6. Carry/Leading Zero correction constraint

Table (3.4) describe the stage of Carry/Leading Zeroes correction for intermediate results:

Table 3.4 Carry/Leading Zero correction of the intermediate result after addition

```
1   constraint carry_leading_zero_correction {
2         if (C) begin
3           {IC_Sig,IC_G,IC_R,IC_S}=={I_Sig,I_G,I_R,I_S}>> 1);
4           (IC_Exp == I_Exp + 1);
5         end
6         else if (I_Sig[`p-1]) begin
7           {IC_Sig,IC_G,IC_R,IC_S} == {I_Sig,I_G,I_R,I_S};
8           (IC_Exp == I_Exp);
9         end
10        else begin
11          SV == lead_zero_fn ({I_Sig,I_G,I_R,I_S});
```

```
12          if (SV >= I_Exp) C_SV == I_Exp;
13          else C_SV == SV;
14          {IC_Sig,IC_G,IC_R,IC_S}=={I_Sig,I_G,I_R,I_S} <<C_SV);
15          IC_Exp == I_Exp - C_SV;
16        end
17  }
```

After addition/subtraction, the intermediate result is composed of carry (C), significant (I_Sig), exponent (I_Exp), Guard bit (I_G), Round bit (I_R) and Sticky bits (I_S).The intermediate result needs to be corrected if there is a carry or leading zeroes in intermediate significand. When C is 1, which is only valid in addition, the intermediate result is shifted right one place to form the corrected intermediate result (IC_Sig, IC_Exp, IC_G, IC_R, IC_S) and the exponent is incremented by one (lines 2 to 5). No correction is required if C is 0 and the most significant digit of I_Sig is 1 (lines 6 to 9). If the effective operation is subtraction, one leading zero exists in I_Sig if the exponent difference between the two operand is larger than 1, also, more than one leading zeroes exist if the exponent difference is 0 or 1. A shift value (SV) is calculated by counting the leading zeroes and is corrected to not exceed I_Exp (lines 11 to 13). The intermediate result is corrected accordingly by left shift of I_Sig, I_G, I_R and I_S by C_SV as well as decrementing I_Exp by same value (lines 14 to 15).

3.1.7. Rounding constraints

Given the constraints set on the corrected Intermediate significand (IC_Sig), Guard (IC_G), Round (IC_R) and Sticky (IC_S) and the correction constraints in the previous step, now it is time to set constraints to figure out the round value based on the rounding direction. Following is the constraints on round value (RV) based on the rounding direction:

- Round tie to nearest even:

  $$\boldsymbol{RV == IC\_G\ and\ (IC\_R\ or\ IC\_S\ or\ IC\_Sig[0])}$$

  The theory behind this, is that when there is a tie between the two nearest value (Guard bit is 1, Round bit is 0, and Sticky bit is 0), if the intermediate result significand is even (right most bit is 0), the round value should be 0, if the intermediate result is odd (right most bit is 1), the round value should be 1. If either of the Round or Sticky bits is 1, this means that the intermediate magnitude is closer to the larger magnitude and hence the round value is 1.
- Round towards zero:

$$\boldsymbol{RV == 0}$$

This is easily explained since the expected magnitude after rounding is always expected less than or equal to the result before rounding.

- Round towards positive infinity:

$$\boldsymbol{RV == !Sign\ and\ (IC\_G\ or\ IC\_R\ or\ IC\_S)}$$

If the intermediate result is positive (Sign is 0), the magnitude of the rounded result is greater than or equal to the intermediate unbounded result, hence if the guard, round or sticky has 1, the round value should be 1.

- Round towards negative infinity:

$$\boldsymbol{RV == Sign\ and\ (IC\_G\ or\ IC\_R\ or\ IC\_S)}$$

If the intermediate result is negative (Sign is 1), the magnitude of the rounded result is greater than or equal to the intermediate unbounded result, and hence if the guard, round or sticky has 1, the round value should be 1.

After the rounding value is constraint, a further addition step to the intermediate significand with the rounding value, is following by a further normalization for carry after rounding is required. Table (3.5) explains the constraints for this, if there exist a carry after rounding, the exponent should be incremented and the round significand should be shifter right by 1 (lines 5 to 8), otherwise nothing should be done (line 9 to 12).

Table 3.5 Addition/Normalization due to rounding

```
1  constraint addition_after_round {
2    {roundCarry,roundSignificand} == ({1'b0,IC_Sig} + RV);
3  }
4  constraint normalization_after_rounding {
5    if (roundCarry == 1'b1)  begin
6      roundNormalizedExponent == IC_Exp + 1;
7      roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
8    end
9    else begin
10     roundNormalizedExponent == IC_Exp;
11     roundNormalizedSignificand == roundSignificand;
12   end
13 }
```

3.1.8. Exception handling constraints

These type of constraints exist to define the behavior of the flags, and the result calculation based on these flags. Below is the equations used for flag calculations:

- Overflow

$$\boldsymbol{overflow} == (\boldsymbol{IC\_Exp} == \mathbf{2}^{\boldsymbol{w}} - \mathbf{1}) \; \boldsymbol{or} \; (\boldsymbol{roundNormalizedExponent} == \mathbf{2}^{\boldsymbol{w}} - \mathbf{1})$$

  The above equation states that overflow flag happens when the exponent value exceeds the maximum value of the exponent which is $2 \times emax$ which is the same as $2^w - 2$. The check on exponent is done after the normalization stages of the intermediate results or the rounded results.

- Underflow

  When shift value required is greater than the exponent value of the intermediate result, underflow flag is raised. This means that the value in the intermediate result cannot be expressed unless it crosses the lower boundary of the possible values of the intermediate significand.

- Inexact

  Inexact happens when the intermediate guard, round or sticky is 1, or when overflow or underflow happens.

- Invalid

  Only happens for the following scenarios:

  - An operand as sNaN,
  - subtraction of infinities, $(+\infty) \mp (\pm\infty)$,

The flow chart in Figure (3.6) explains the constraints on the final result if one of the above flags are raised; if the invalid flag is raised, the output should be NaN, else if any of the operands is infinity the result should be infinity, else if the overflow flag is not raised, the result should be the same as the rounded normalized result, else if overflow flag is raised, depending on the round direction the final result is calculated to be either the maximum value of floating point or $\pm\infty$.

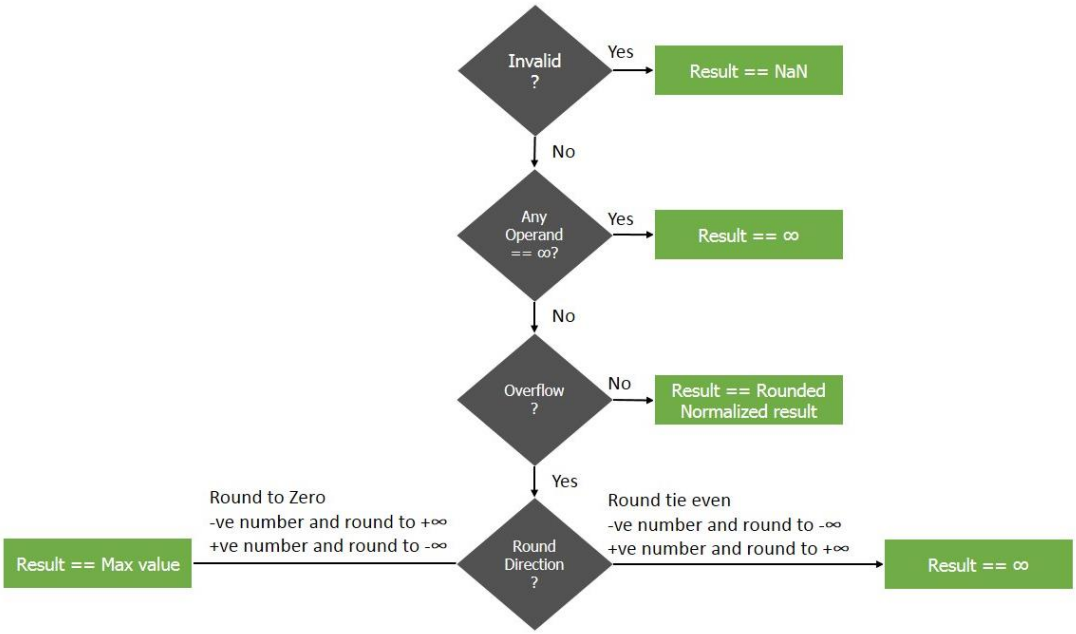

Figure 3.6 Result constraint due to exceptions

3.2. Multiplication

Figure (3.7) summarizes the data path for Operation constraints for Multiplication. No need to choose the higher/lower operands and normalize the lower operand in multiplication, in fact, the multiplication data path is much shorter than the addition.

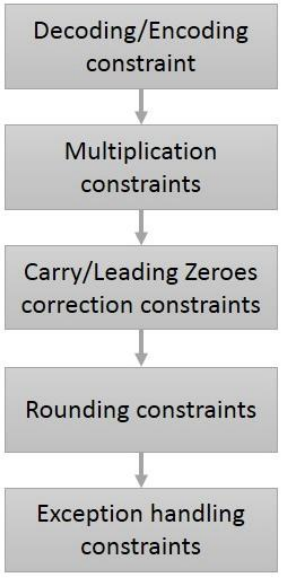

Figure 3.7 Multiplication Operation constraints

3.2.1. Multiplication constraints

Using System Verilog multiply operator (*) was inefficient, it took a lot of time to parse this constraint and slow down the generation of test vectors dramatically. Hence, we deploy partial product summation of both operands through creating a function that shift the first operand to the left and depending on the value of the i^{th}element of the second operand, this partial product will be added or not; if the value of the i^{th}element in operand2 significand is 0, it is skipped, else, it is added. Figure (3.8) explains the layout of the partial product summation, and Table (3.6) shows the SV function to implement this algorithm.

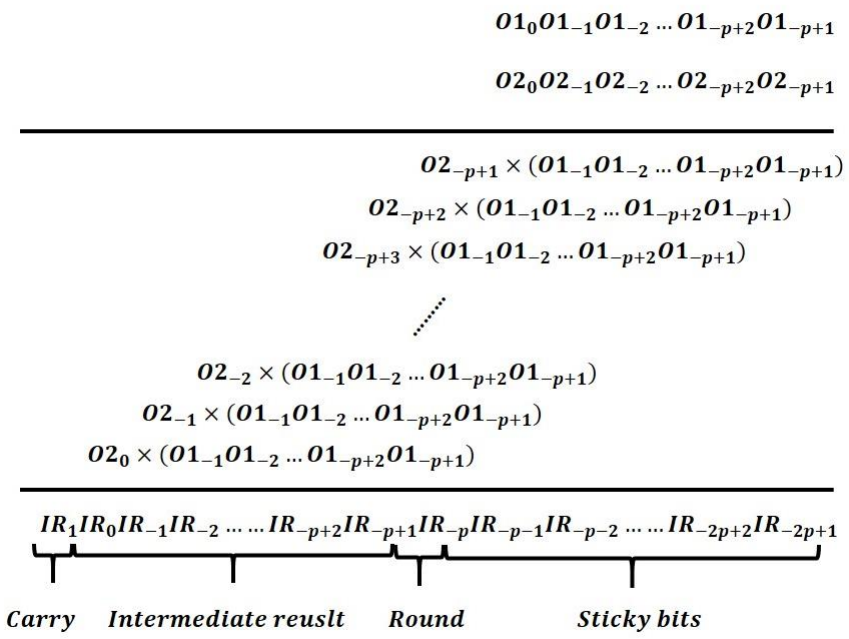

Figure 3.8 Partial products summation

As seen in Figure (3.8), the intermediate result is twice the precision of the operands to calculate the Round and Sticky; the Round is the first bit to the right of the bits taken in the intermediate significand, and the Sticky is disjunction all the other bits. The sign is constraint to be the exclusive or of the operands' signs, the exponent is constraint to be the summation of both exponents while subtracting the Bias value with is equal to $emax$.

Table 3.6 constraint function for partial produce summation

```
1   function [2*`p-1:0] partial_product_multiplication(input [`p-1:0]
2   op1,op2);
3       bit [2*`p-1:0] normal[0:`p-1], shifted[0:`p];
4       shifted[0] = 0;
5       for (int i = 0; i <= `p-1; i++)
6       begin
7           if (op2[i])
8               normal[i] = {`p'b0,op1};
9           else
10              normal[i] = '0;
11          shifted[i+1] = (normal[i] << i) + shifted[i];
12      end
13      return shifted[`p];
    endfunction
```

3.2.2. Carry/leading Zeroes correction constraints

Same as section 3.1.6

3.2.3. Rounding constraints

The RV equations slightly differ from section 3.1.7, as in multiplication, there is no Guard bit, only Round and sticky, and hence the equations are:

- Round tie to nearest even:

$$\boldsymbol{RV == IC\_R\ and\ (IC\_S\ or\ IC\_Sig[0])}$$

- Round towards zero:

$$\boldsymbol{RV == 0}$$

- Round towards positive infinity:

$$\boldsymbol{RV = !Sign\ and\ (IC\_R\ or\ IC\_S)}$$

- Round towards negative infinity:

$$\boldsymbol{RV = Sign\ and\ (IC\_R\ or\ IC\_S)}$$

3.2.4. Exception handling constraints

Overflow, underflow and inexact exceptions are the same as computed in section 3.1.8, while for invalid flag, the following is accounted for:

- An operand as sNaN,
- $0 \times \infty, \infty \times 0$,

3.3. Division operation constraints

Figure (3.9) summarizes the data path for the Operation constraints for Division operation. Again, no need to choose the higher/lower operands, yet if the divisor is subnormal, a normalization step should be done before going through the division algorithm in the division constraint section.

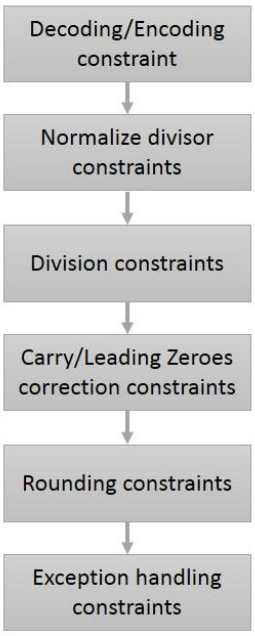

Figure 3.9 Division Operation constraints

3.3.1. Normalize divisor constraint

These constraints act as a filter to divisor; if the divisor is normal number, it passes without modification, else, it requires normalization by shifting it to the left and saving the shift value later for exponent correction of the intermediate result.

Table 3.7 Normalize divisor constraints

```
1  function int leading_zero_calculation (input [0:`p-1] functionSignificand);
2      for (int i = 0;i < `p;i++) begin
3          if (functionSignificand[i] == 1'b1) return i;
4      end
5  endfunction
6  constraint divisor_normalized {
7      exponent_correction == leading_zero_calculation(divisorSignificand);
8      divisorNormalizedSignificand == divisorSignificand <<
9  exponent_correction;
   }
```

As shown in Table (3.7), a function is implemented to calculate the leading zeroes in the divisor, this will return 0 if the divisor is normal, and will return the number of leading zeroes if it is subnormal. The divisor significand is normalized by shifting the leading zeroes out, and an exponent correction variable is saved for later update of the intermediate result exponent.

3.3.2. Division constraints

Again, using built in System Verilog operator of division "/" shows slow response in generation of the test vectors, and hence an iterative restoring algorithm for division was carried out.

The algorithm works with 3 variables and a comparator, these variables are:

- The quotient register: it is twice the precision of the binary encoding format,
- The divisor register: it is three time the precision of the binary encoding format,
- The remainder register: it is three time the precision of the binary encoding format.

The following are the steps used in the iterative algorithm:

1- Initializing the registers as shown in Figure (3.10):

a. The quotient register is initialized with all 0's,
b. The divisor register is initialized with the divisor significand aligned to the left,
c. The remainder register is initialized with the dividend significand aligned to the left,

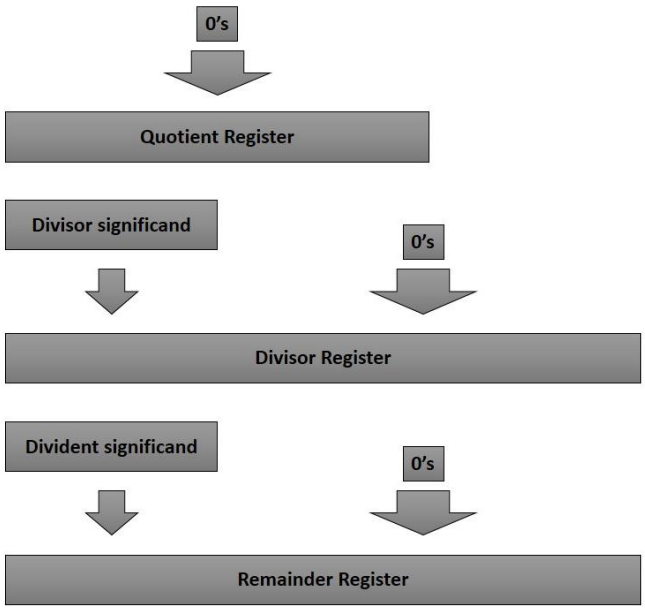

Figure 3.10 Initial step for iterative division

2- Compare the remainder resister with the divisor register
   a. If the remainder is greater than or equal to the divisor register:
      i. Update the remainder register by subtracting the value in the divisor register
      ii. Insert 1 to the right most bit of the quotient and shift left its content
   b. If the remainder is less than the divisor register:
      i. Restore the remainder with its old value
      ii. Insert 0 to the right most bit of the quotient and shift left its content
3- Shift right the divisor register by 1 bit place
4- Repeat steps 2 and 3 for 2*precision time, (until all the quotient bits are written in)

Table (3.8) shows the SV function that implements this restoring iterative division algorithm, also, the implementation for the iteration is in Figure (3.11).

Table 3.8 SV function to implement the iterative restoring division algorithm

```
1   function [`p-1:-`m] iterative_div(input [`p-1:0] divident,divisor);
2           bit [2*`p-1:-`m] r[-`p+1:`p], d[-`p+1:`p];
3           bit [`p-1:-`m] q[-`p+1:`p];
4           r[-`p+1][2*`p-1:`p] = divident;
5           q[-`p+1] = '0;
6           d[-`p+1][2*`p-1:`p] = divisor;
7           for (int i = -`p+2; i <= `p; i++)
8           begin
9               if (r[i-1] >= d[i-1])
10              begin
11                  r[i] = r[i-1] - d[i-1];
12                  q[i] = {q[i-1][`p-2:-`m],1'b1};
13              end
14              else
15              begin
16                  r[i] = r[i-1];
17                  q[i] = {q[i-1][`p-2:-`m],1'b0};
18              end
19              d[i] = {1'b0,d[i-1][2*`p-1:-`m+1]};
20          end
21          return q[`p];
22  endfunction
```

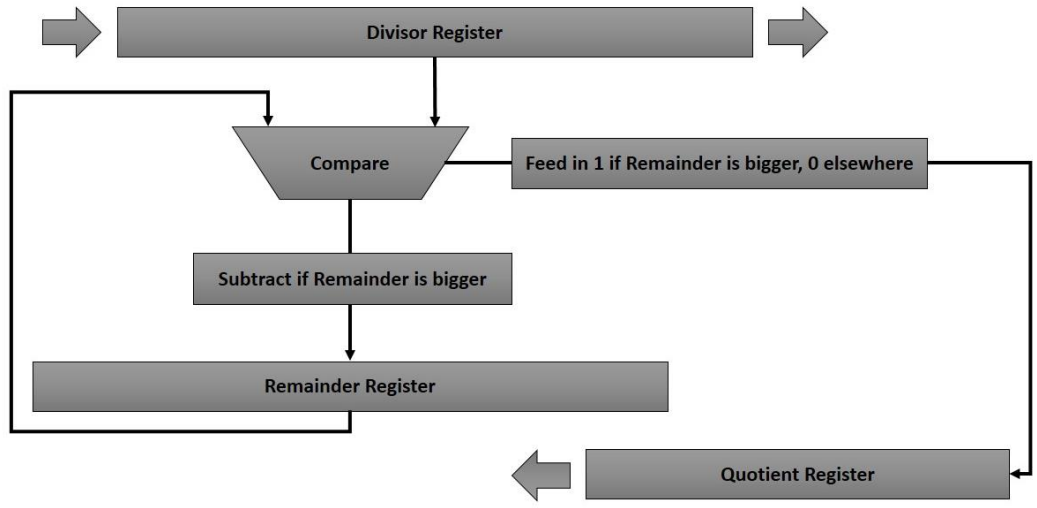

Figure 3.11 the iterative, restoring division algorithm

As mentioned earlier, the quotient register is twice the precision with, this is to account for the fact that the intermediate result before rounding should be unbounded. Figure (3.12) shows the mapping between the 2*p wide quotient and the intermediate significand, guard, round and sticky bits.

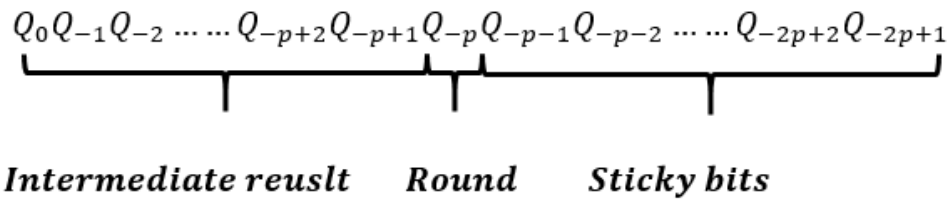

Figure 3.12 Mapping between quotient and intermediate result in division

The sign of the intermediate result is the exclusive or of the divisor and the dividend signs, the exponent is the same as that of the dividend.

3.3.3. Carry/Leading zeroes correction

In division, there is no carry, and since there was a step earlier to align the divisor to the dividend for subnormal divisors, here there is only one correction that needs to be handled, it is correcting the exponent of the intermediate result by adding the exponent correction computed in Table (3.7).

3.3.4. Exception handling constraints

For overflow, underflow and inexact flags, they are the same as in addition/subtraction or multiplication, but for the invalid flag, it is different, also there exist a new flag with is divide by zero, below is how both flags are constraint

- Invalid:
  - $\frac{0}{0}$
  - $\frac{\infty}{\infty}$

- Divide by zero:
    - $\frac{x}{0}$, where x in a finite floating point number, the result's sign is exclusive or of the dividend sign and the divisor sign,

3.4. Fused multiply Add constraints

Fused multiple add/subtract is a multiplication followed by an addition or subtraction operation. Figure (3.13) shows the constraints on the data path through an FMA operation. The operation differ from multiplication followed by addition/subtraction in the following points:

- The significand of the result of multiplication is expected to be unbounded, and hence the addition/subtraction operand width is twice the precision of the addition/subtraction discussed in section 3.1.
- The invalid flag is raised when:
    - An operand as sNaN,
    - $(0 \times \infty) + c, (\infty \times 0) + c$

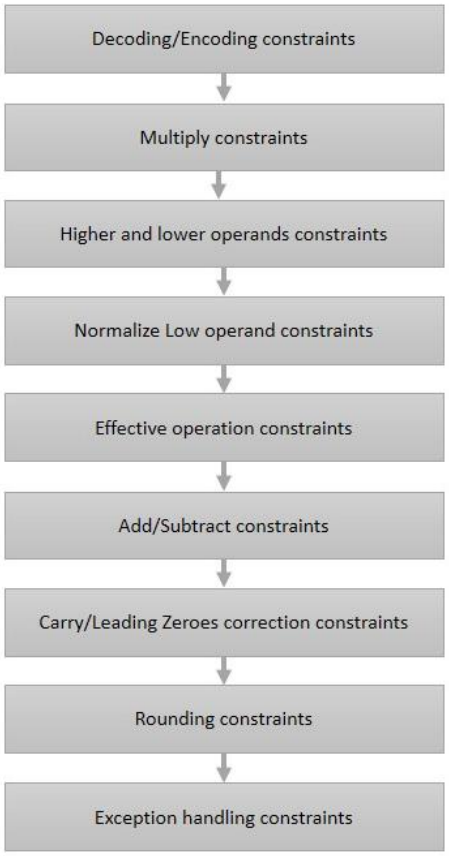

Figure 3.13 FMA operation constraints

3.5. Square Root constraints

Figure (3.14) summarizes the data path for the Operation constraints for Square root operation. In square root operation, we only deal with one operand, we assume that this operand is positive. Square root for negative numbers is not covered in our work. if the operand is subnormal, a pre-normalization step should be done before going through the square root algorithm, then a post normalization step restores the result to account for the the operand sub normality.

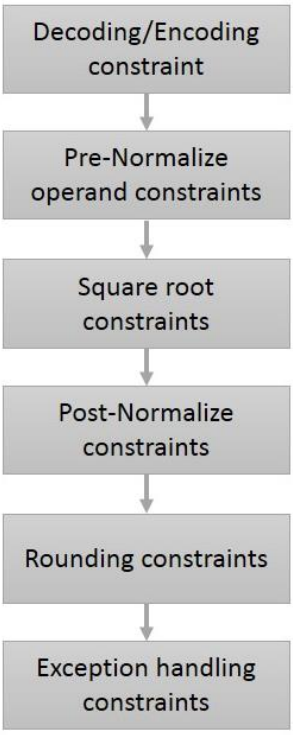

Figure 3.14 Square Root Operation Constraints

3.5.1. Decoding/Encoding constraints

Same as in section 3.1.1, but only for one operand and assumption that the sign bit is always zero (positive).

3.5.2. Pre-Normalization step

This step is concerned with converting subnormal representation to a normal version by shifting left the operand significand until the most significand digit is 1, to be able to utilize the square root algorithm explained later that only deals with normal numbers. Table (3.9) explains the SV constraints for this normalization step.

Table 3.9 SV constraint to implement Pre-normalization step for subnormal numbers

```
1  function int ShiftValue (input [`p-1:0] i1);
2          int j;
3          ShiftValue = 0;
4          for (int i = `p-2; i >=0 ; i --)
5          begin
6              j = (`p-1) -i;
```

```
7                   if (i1[i] && ShiftValue == 0)
8                       ShiftValue = j;
9               end
10  endfunction
11  constraint pre_norm {
12    SV == ShiftValue(operand1Significand);
13    intermediateSignificand == operand1Significand << SV;
14    intermediateExtendedSignificand == {intermediateSignificand,`p'b0,2'b0};
15  }
```

Another thing that is handled in this step extending the significand from the right with zero padding. This step is important since the result of the square root is always half the precision of the operand under the root. So to return the same precision as the input to the square root algorithm, this extension is required. Also, another two bits are extended so that the square root algorithm return one more bit on the right that is mapped to the round bit. Line 14 in Table (3.9) denotes the right padding step.

3.5.3. Square Root constraints

Two different approaches were carried in calculating the square root for a given floating point binary number. The approaches share the calculation of the exponent.

If the biased exponent is odd, subtracting the bias value which is equal to maximum possible exponent (Odd number), will lead to an even number, then the result exponent is calculated to be half the unbiased even number, next the bias is added one more time.

If the biased exponent is even, subtracting the biased value will lead to an odd number, then the result exponent is calculated by subtracting one from the exponent to make it an even number then halving that number, then adding the bias one more time.

The following equation summarize how the result exponent is calculated, where RE and OE are result exponent and Operand exponent respectively:

$$\boldsymbol{Half\_bias} = (\boldsymbol{Emax} - \mathbf{1})/\mathbf{2}$$

$$\boldsymbol{RE} = \boldsymbol{OE}[\boldsymbol{W} - \mathbf{1}:\mathbf{1}] + \boldsymbol{Half\_bias} + \boldsymbol{OE}[\mathbf{0}]$$

To account for the operands with even exponents, a correction step is required to the significand by left shift the significand by one. Hence, the fixed point representation of the number under the root is either 01.XXX or 1X.XXX, which is a number between 1.000 and 3.999 and hence, the result of the square root is always between 1.000 and

1.999 and hence the square root algorithm will always return 1.YYY, so there is no need to worry about normalization after the square root is done for normal numbers.

3.5.3.1. First Square root algorithm

The first algorithm is straight forward algorithm using the capabilities of SV language; i.e. using the SV power operator to calculate the square root by having the power equals 0.5, then the remainder is calculated by squaring the result and subtracting it from the original number as shows in Table (3.10). The Round bit is the right most bit of the Quotient Q, and the Sticky bit is the disjunction of the Remainder R.

Table 3.10 First approach in calculating the square root using SV power operator

```
1  function [`p+1:0] sqrt_approach_1(input [`p+1:0] i1);
2          logic [`p:0] R,Q;
3          logic Sticky;
4          Q = {i1,`p'b0,1'b0} ** 0.5;
5          R  = i1 - (Q ** 2);
6          Sticky = | R;
7          return {Q,Sticky};
8  endfunction
```

3.5.3.2. Second Square root algorithm

The second approach is based on the proposal in [43], it is an iterative approach based on the algorithm explained in Table (3.11). The algorism depends on having three registers, each is two more bit wide than the operand significand

1- Quotient register (Q): will contain the result significand of square root, and the Round bit after the iterations are done
2- Factor register (F)
3- Remainder register (R): after end of iterations, if it contain non zero value, it means that the sticky bit is one, else, sticky bit is zero

Figure (3.15) explains how the above registers are initialized, both Quotient and Factor registers are initialized with zeroes while the Remainder register is initialized with Radicand (operand) most 2 significand digits from the right, and zeroes elsewhere.

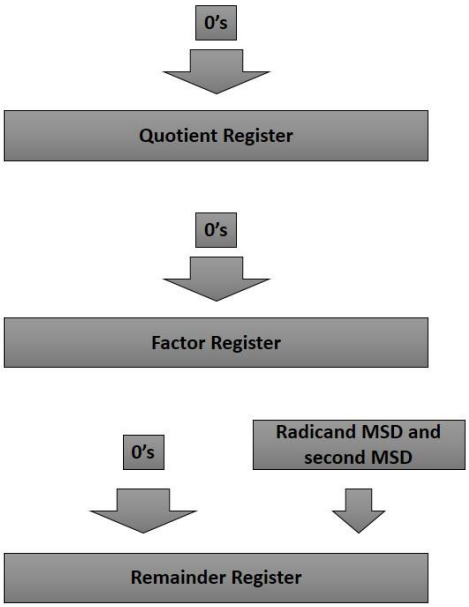

Figure 3.15 Registers initialization for the iterative approach

Table 3.11 First approach in calculating the square root using SV power operator

1	Initialize radicand with input data value, R=0, Q=0, F=0, i=n; n is MSB bit-index of Radicand (D).
2	Divide the radicand into sub-groups which each sub-group consists of 2 digits starting from integer LSB.
3	Start the calculation from MSB sub-group to LSB sub-group. Treat current sub-group as current partial remainder. Rt = Dt[i:i-1]; t is time index indicator.
4	Do a comparison whether current partial remainder is bigger or equal than current partial factor ((Ft<<1)|1), (shift left of factor register, with one entering from the right) If yes Update Q; Qt+1 = (Qt<<1)|1; Update F; Ft+1 = ((Ft+Ft[0])<<1)|1; Else Update Q; Qt+1 = (Qt<<1)|0; Update F; Ft+1 = ((Ft+Ft[0])<<1)|0;
5	Do subtraction to partial remainder by the result value of factor multiplication; Then append the subtraction result with next subgroup data of Radicand in the LSB position of partial remainder, in order to update R. Rt+1 = ((Rt–(Ft*Ft[0]))<<2)|D[i-2:i-3];
6	Update the current indexes for next use. t+1 is changed into t; i-2 is changed into i; i-3 is changed into i-1;
7	If the process is not over Jump to step 4 and loop the process.

	Else Latest Q value is final square root value. Latest R value is final remainder value.

Figure (3.16) is a tutorial example explaining the above iterative approach to calculate the square root of 445.

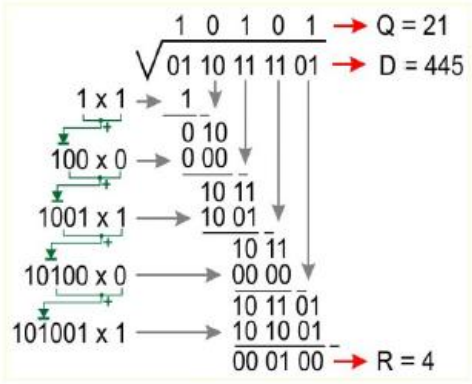

Figure 3.16 Tutorial example of the iterative approach

Since 445 is represented in odd number of bits in binary "110111101", we add a dummy zero bit from the left to be "0110111101", and hence we have 5 subgroups from the radicand. The first subgroup is "01", we initialize the remainder register to be "01", then we compute ((Ft<<1)|1) as explained in step 4, to be "1", so remainder register is greater than or equal to the computed factor, and hence we should update the Quotient and Factor register as explained in step 4, so that the Quotient is left shifted with one to be "1" and the new factor register is calculated by ((Ft+Ft[0])<<1)|1), to be "1". We calculate the new remainder by computing (Rt+1 = ((Rt–(Ft*Ft[0]))<<2)|D[i-2:i-3]) as explained in step 5, and hence the new remainder is "010". This is the end of the first iteration.

Again, in the next iteration, we compare the current remainder "010" with ((Ft<<1|1) which is "11", and hence we find that it is not greater than or equal the remainder, and hence we execute the else part in step 4, so the new Quotient is "10" and the new factor is "100". We use the equation in step 5 to compute the new remainder which will be "1011". This is the end of the second iteration.

We repeat the above until the last subgroup of the Radicand enters the remainder, and then as shown in Figure (3.16), the Quotient register will contain "10101" which is

equivalent to 21, and the remainder register is not empty “000100”, hence the value is approximated. This is clear as square root of 445 is 21.09502310972899.

As mentioned in the pre-normalization section, for both approaches, the radicand to undergo the square root is padded with zeroes to:

1- Create a Quotient with same precision as the radicand, hence the radicand is padded with p zero digits from the right,
2- Two more zero digits are added from the right to create one more bit in the Quotient to account for the Round bit that will be used in rounding step later

As for the Sticky bit, it is calculating from the disjunction of the remainder register.

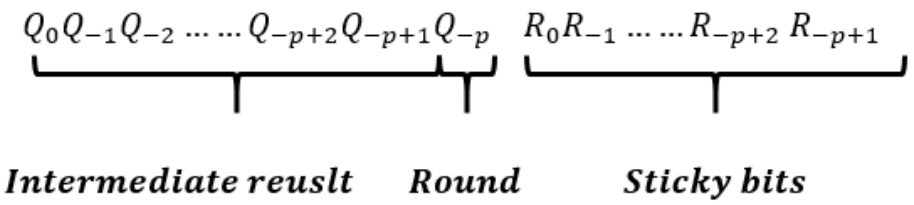

Figure 3.17 Mapping between quotient and intermediate result in square root

3.5.4. Post normalization step

If the Operand under the square root was subnormal, there existed a pre-normalization step that made a left shift version of the significand, in this stage, we correct this unforsaken left shift by a half right shift of the Quotient significand.

3.5.5. Rounding constraints

After calculation the Round, Sticky bits, depending on the round direction, the final significand of the square root operation is calculated, by the same equations in 3.2.4.

3.5.6. Exception handling

Given the assumption that the operand is positive, only one exception needs handling, which is the having a NaN under the square root, and it should result to NaN. Square root operation will never return overflow or underflow since the result exponent is always less in magnitude than the operand exponent.

3.6. User defined constraints

As mentioned in the proposed algorithm, besides having constraints covering the data path for every operation, the user is allowed to add his own constraints to hit desired corner cases. Table (3.12) is an example of constraints set during addition/subtraction to hit a corner case that will be shown later to cover a bug in one floating point unit design. The user constraints add more restrictions on the effective operation to be subtraction and intermediate results to have one leading zero.

Table 3.12 Exmaple of User defined constraints

```
1  constraint intermediateSignificand_userConstraints {
2          effectiveOperation == SUB;
3          I_Sig[`p-1:`p-2] == 2'b01;
4  }
```

Chapter 4 Results and comparisons

In this chapter we discuss the result of our proposal and draw comparisons with other verification algorithms for floating point units. All our verification were applied on machine with the following specifications:

- Operating system: CentOS 6.3,
- Platform: 64 bits,
- Ram: 48GB,
- Cores: it has 16 cores, each with frequency of 2.8 GHz

We used QuestaSim Tool in all our simulation, with the following version:

- QuestaSim-64 10.4c_1 Compiler 2015.09 Sep 4 2015

4.1. Advantages of Our proposal

4.1.1. No solver, No modelling

One advantage of the proposed algorithm, is that it does not require bit level analysis or creating Cartesian equations and developing a particular engine to solve these equations as in [2], [3], [5], [11], [12].

4.1.2. Based on System Verilog Language

The algorithm is a plug and play utility that can be simulated with any simulator that supports SV constraints. Since most Hardware verification nowadays are based on SV language, the proposal doesn't require pre-learning for most of verification engineers.

4.1.3. Global solution for verification

Since the methodology is to create test vectors, these test vectors can be applied at any time of the product life cycle of a floating point unit, it can be used within behavioral simulation with C or Matlab, it can be applied to the RTL and verify its output, it can be converted to floating point instructions and test any current processor having floating point hardware.

4.1.4. Fast generation of test vectors

Recently, verification is the bottle neck in any software/hardware design life cycle due to the huge time and effort spent versus the fast time to market requirements. Our methodology shows promising figures with respect to time when being simulated with QuestaSim simulator.

Figure (4.1) shows the average time to generate 1 test vector for addition, subtraction, multiplication, division and square root with single and double precision binary formats. As shown, the average time to generate one test vector in multiplication with single precision is 0.5 millisecond, the average time to generate one test vector in multiplication with double precision is 2.5 milliseconds. For Square root, the average time to generate one test vector in single precision is 0.9 milliseconds, and the average time for double precision is 1.3 milliseconds. For division, the average time to generate one test vector with single precision is 2.4 milliseconds and the average time to generate one test vector with double precision is 2.7 milliseconds. Division is a bit slower than multiplication and square root which is reasonable due to the iterative approach deployed in the constraints for division, and due to the extra step of normalization when dealing with subnormal divisor. Addition and subtraction are the slowest, with 11.7 milliseconds to generate one test vector in single precision and 22.8 milliseconds in double precision, again this is justifiable by the fact that the constraint data path for addition and subtraction is longer than that of multiplication and division, since it requires computing the effective operation, arranging the operands to identify what is the higher and the lower operands and finally normalization of the lower operand.

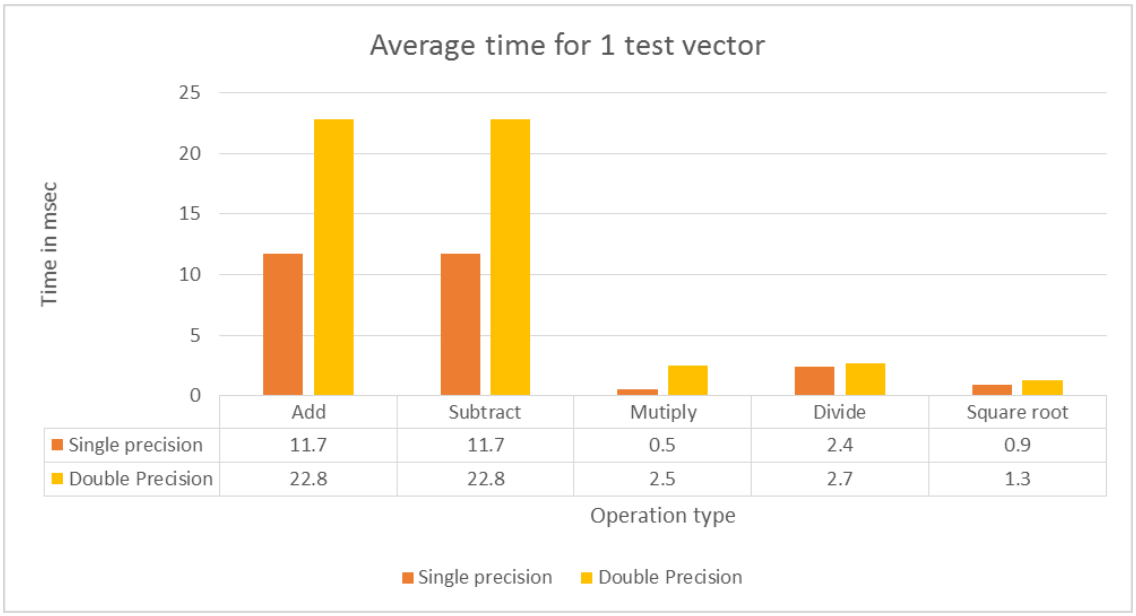

Figure 4.1 Average time to generate 1 test vector for different operations

4.1.5. Linear response with respect to required number of test vectors

As shown in Figures (4.2,4.3,4.4,4.5), increasing number of test vectors (N) shows linear response in time, which provide good scalability for our proposal.

Yet increasing number of cores as shown in Figures (4.6,4.7) doesn't significantly improve time. This is due to the fact that running on multicore improve time when the design has multiple hierarchical level, yet our implementation includes all the SV constraints in one hierarchy. The experimental results here are run with single precision on single core, four cores and eight cores.

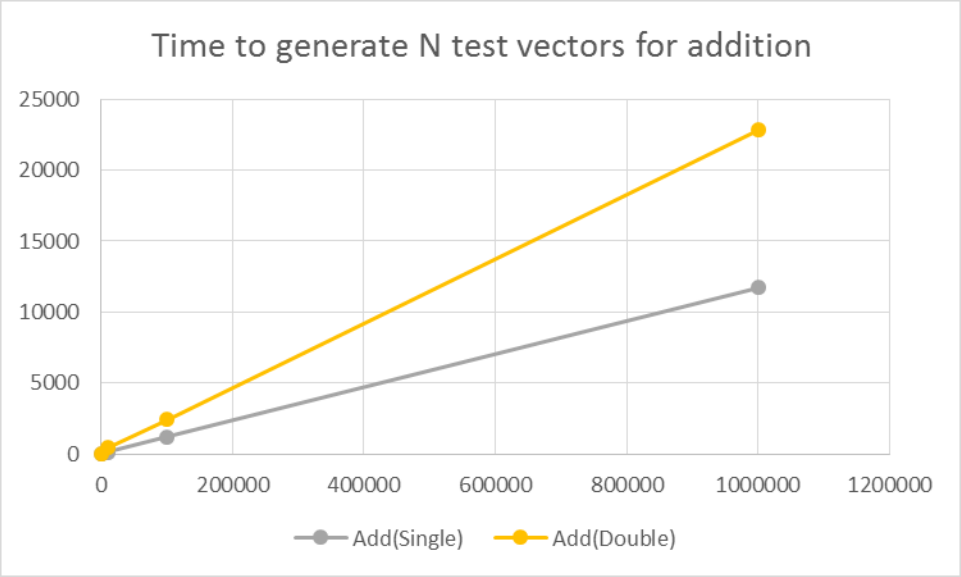

Figure 4.2 Time to generate N test vectors for addition

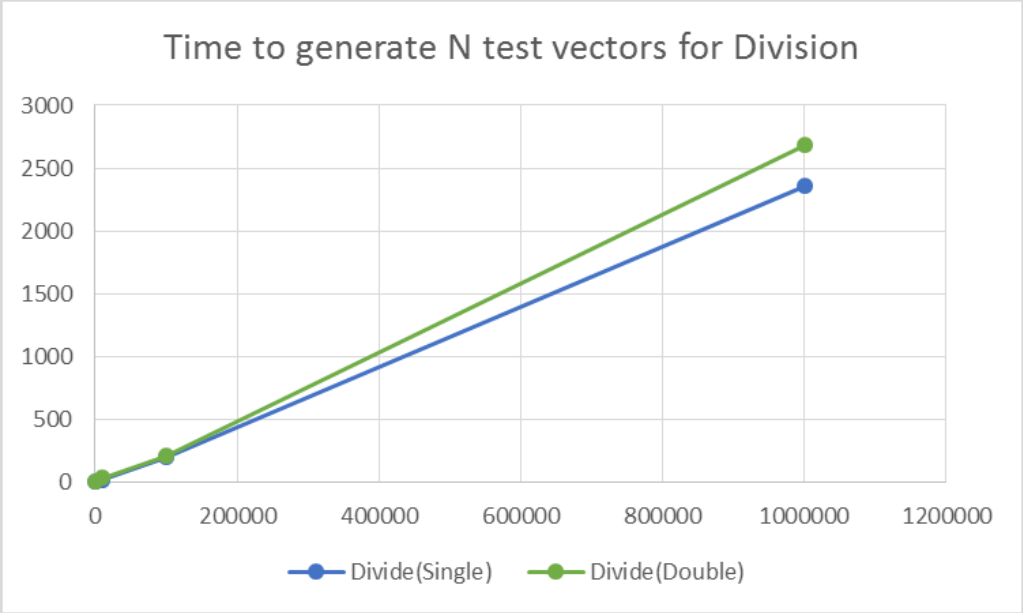

Figure 4.3 Time to generate N test vectors for division

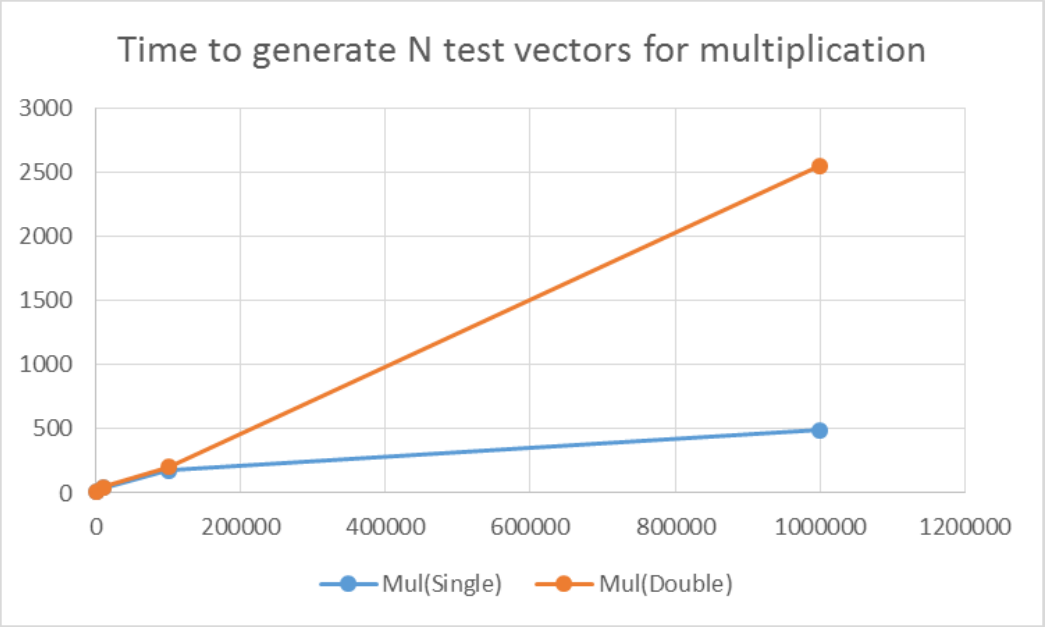

Figure 4.4 Time to generate N test vectors for multiplication

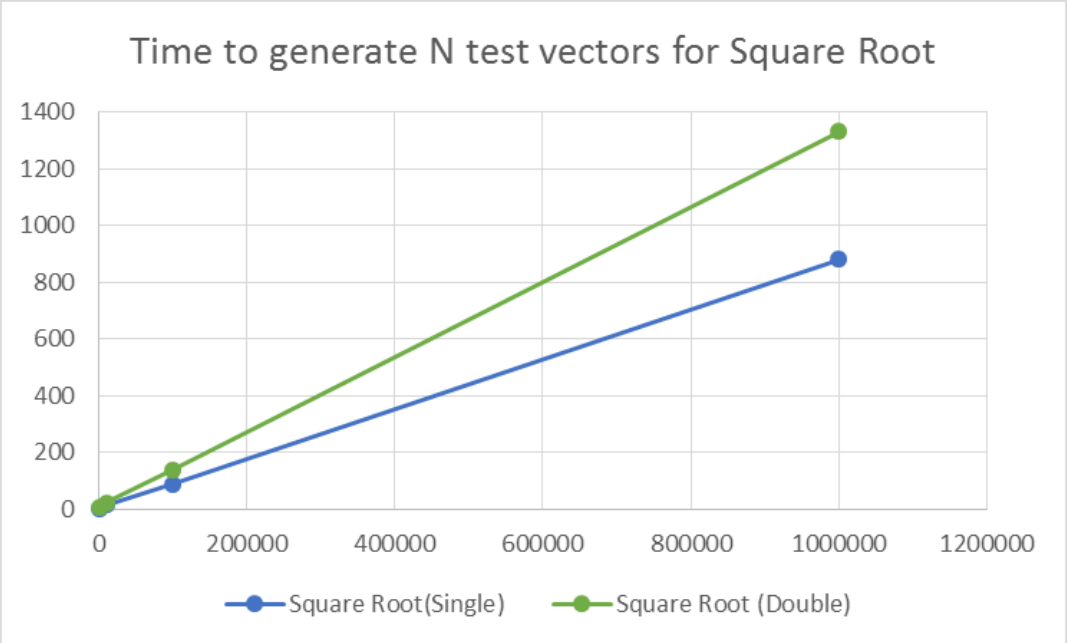

Figure 4.5 Time to generate N test vectors for square root

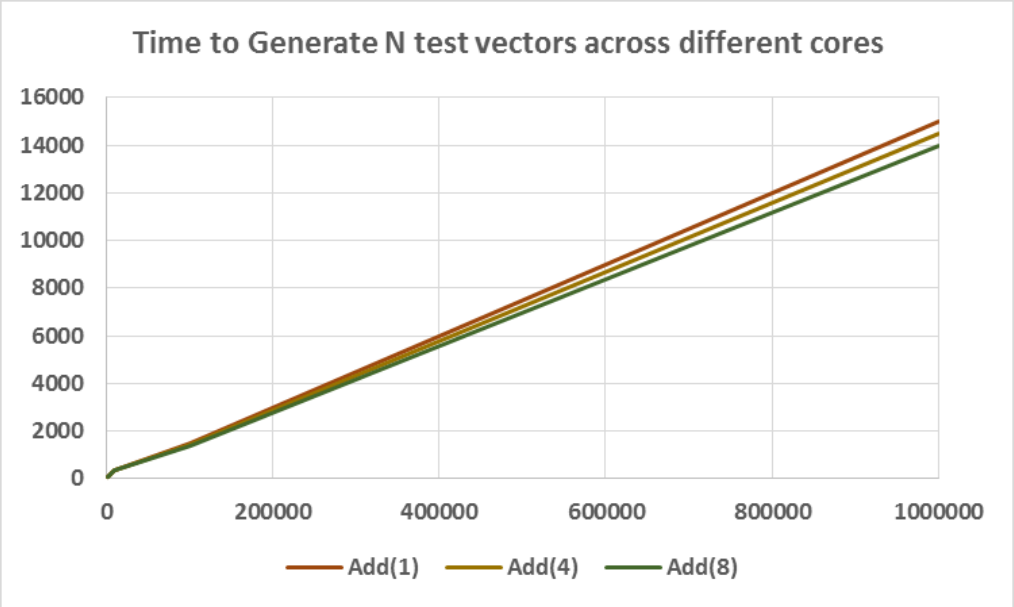

Figure 4.6 Time to generate N test vectors for addition across different cores

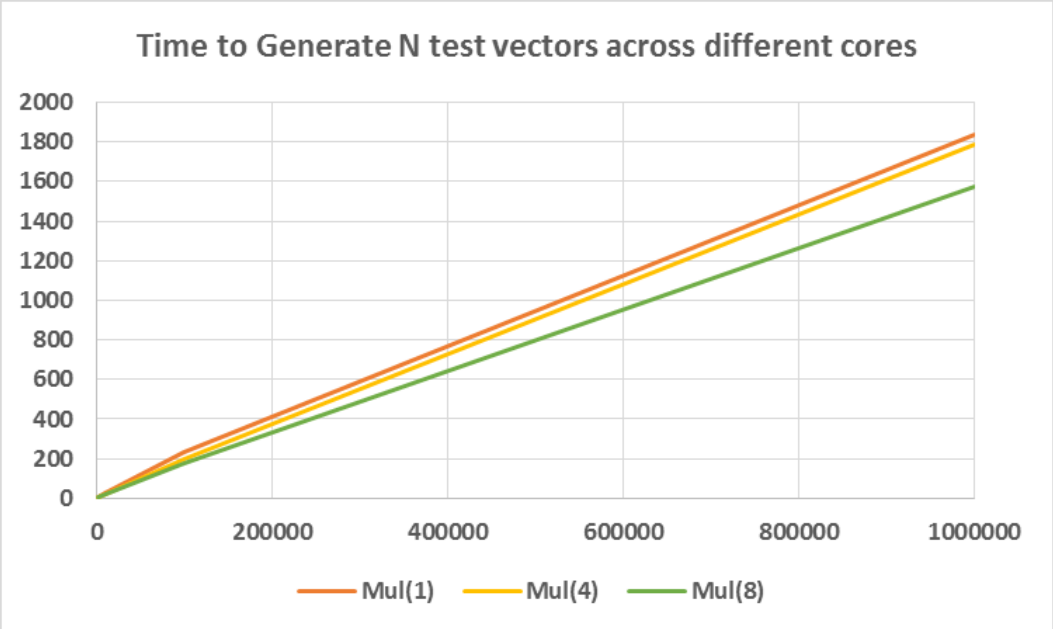

Figure 4.7 Time to generate N test vectors for multiplication across different cores

4.1.6. No scaling issue with bigger precision

If one refer back to Figure (4.1), it is clear that the double precision take more time than that of single precision, yet the increase in time is not huge. Figure (4.8) summarizing the ratio of increase from single to double precision across different operations:

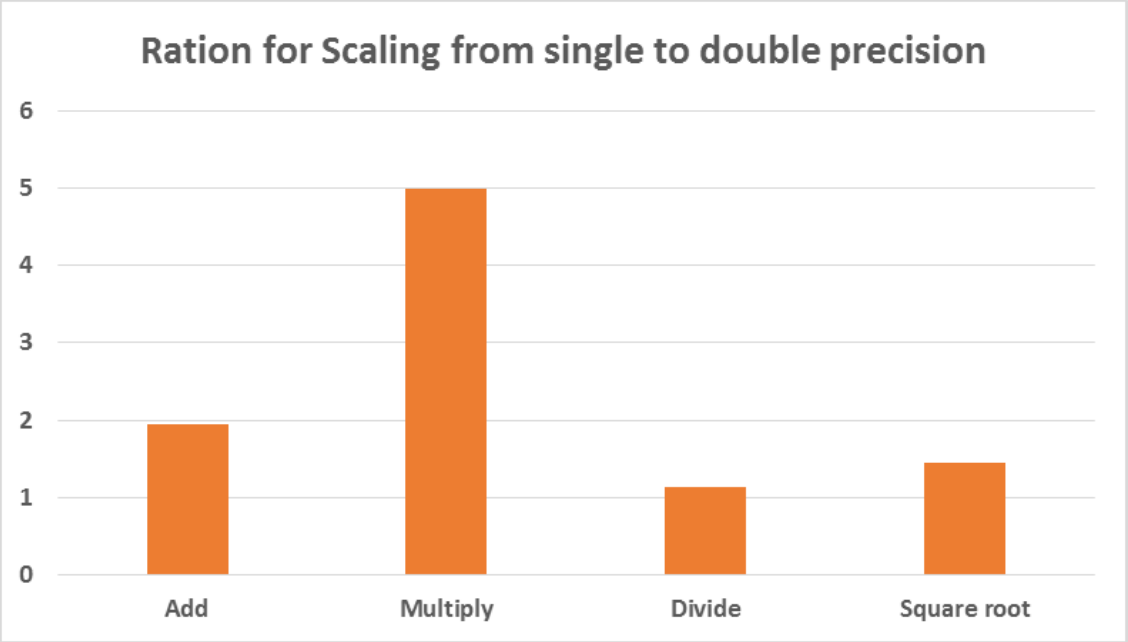

Figure 4.8 Ratio of increase in time from single to double precision

Multiplication shows the worst figures here, again this is justifiable since the partial product width is scaled up from 24 bit wide to 53 bit wide, and instead of addition 24 times, it is done 53 times. For Addition/subtraction, the ratio is almost 2 which is also justifiable since the width of the significand is doubled. Division figures are the best here where the ratio is almost 1, which means that there is no scaling issue at all for the division algorithm applied, this is due to applying an iterative approach in computing the quotient and remainder. Finally, for square root, the ratio is almost 1.5, this can be explained; when referring to the algorithm explained in 3.5.3.1, one will notice that in order to compute the remainder, power and squaring operation are required, which will vary when scaling from single to double precision as the operand's width undergoing power/square increased.

4.2. Comparison with other related work

4.2.1. Comparison with FPgen

The experimental results in [2] implied that it requires 6 minutes to generate 1586 test vectors for a combination of addition, multiplication and division, which means that generation of one test vector takes about 227 milliseconds, while our slowest figures

which are captured during simulation of addition/subtraction constraints take only 22.8 milliseconds, which is at least ten times faster than that of FPgen.

4.2.2. Comparison with decimal floating point constraint solvers

As mentioned in [44], the maximum time to generate one test vector for double precision floating decimal floating point square root operation is 37 seconds. In our experiments, we show that the maximum time to generate one test vector for double precision binary square root floating point operation is 1.3 milliseconds.

4.3. Summary of bugs discovered

Our verification technique proves to be effective in finding bugs. We deployed the generated test vectors on some software/hardware implementations of Binary FP unit to verify addition/subtraction operations with single precision and we discovered the following bugs:

4.3.1. Bugs in FPU100, an open source design

FPU100 is an Opencores VHDL module, IEEE 754-compliant, single precision soft core [13]. It has been tested with 2 million test vectors and the no bugs were detected since 2007, also, it had been hardware proven as it was implemented in a Cyclone I–EP1C6 FPGA chip and was then connected to the Java processor JOP [16] to do some floating-point calculations. Our framework discovered 4 bugs in subtraction and 2 in multiplication, following are the bugs explained:

4.3.1.1. Wrong Inexact exception calculation:

Inexact exception is raised if the result overflows/underflows, or any of the intermediate guard (G), round (R) or sticky bits(S) is 1. When subtracting two FP numbers and the intermediate result is having one leading zero, guard bit is 1 whilst both round and sticky are 0, doing a left shift will clear the guard bit, so the result is expected to be exact. FPU100 raised inexact flag in this scenario. Figure (4.9) shows the intermediate result that causes the bug, since G, R and S are 0’s, the result should be exact.

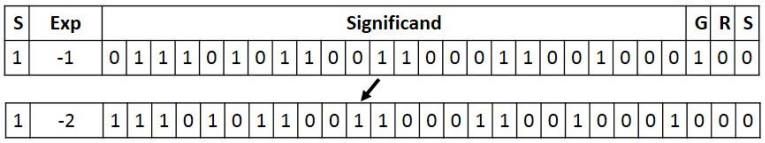

Figure 4.9 Wrong inexact flag with subtraction in FPU100 design

4.3.1.2. Wrong result when two normal numbers are subtracted and return a subnormal number:

When subtracting two normal numbers and the result has leading zeroes that are more than the intermediate exponent, the result is subnormal, and the intermediate result is shifted with a corrected value as explained in section II, Carry/Leading Zeroes code, lines 12 to 14. FPU100 design returns wrong results, which varies with the value of the guard bit; if the guard bit is 0, the result is infinity with both overflow and inexact exceptions set, if the guard bit is 1, the result is zeroes with both underflow and inexact flags set. Figure (4.10) shows the intermediate results that causes the mentioned bugs, comparing the correct results vs FPU100 results.

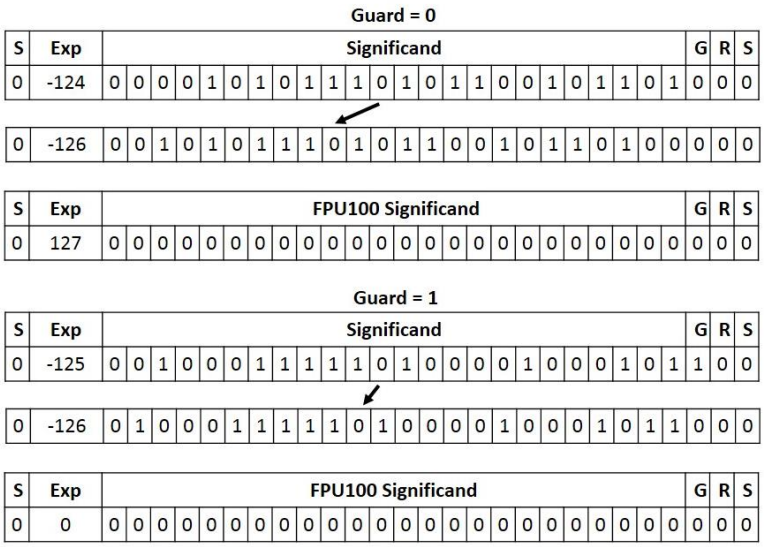

Figure 4.10 Wrong result when two normal numbers are subtracted and return a subnormal number in FPU100 design

4.3.1.3. Subtracting positive zero from negative zero:

For the following: "(-0)-(+0) =?" the result is negative zero (-0), yet the FPU100 returns positive zero (+0).

4.3.1.4. Wrong result with multiplication when result is subnormal and underflow occurs:

When multiplying one normal and one subnormal number, the result can have leading zeroes, if the leading zeroes are equal to the difference between the intermediate exponent and minimum exponent (emin = -126 for single precision), whilst the guard, round and sticky are 1's, underflow exception is raised and the result shall return in subnormal format by left shifting of the significand by a shift value equal to the difference between the two exponents. FPU100 return the intermediate significand un-shifted as shown in Figure (4.11).

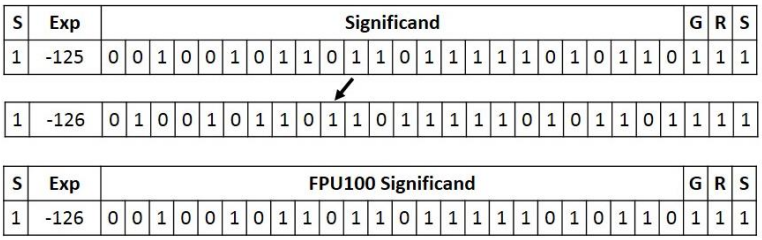

Figure 4.11 Wrong result with multiplication when result is subnormal and underflow occurs in FPU100 design

4.3.1.5. Wrong Output, Inexact and Underflow exceptions with multiplication when underflow occurs:

This is a similar scenario to bug#4, the only difference is that the leading zeroes are bigger than the difference between the intermediate exponent and the minimum exponent. FPU100 shift the result to eliminate the leading zeroes and produce neither underflow, nor inexact exceptions as shown in Figure (4.12).

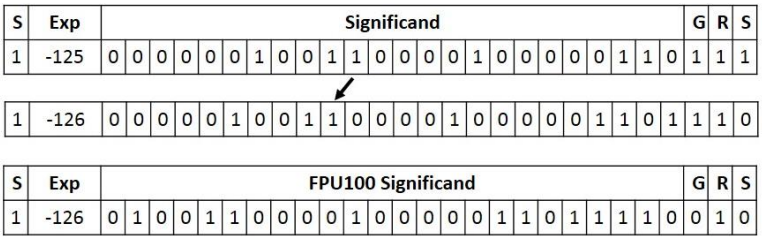

Figure 4.12 Wrong Output, Inexact and Underflow exceptions with multiplication when underflow occurs

4.3.1.6. Wrong result significand with division when the divisor is greater than the dividend

When the divisor is greater than the dividend, the result for FPU100 design is wrong, as seen in Figure (4.13), where the exponent is correct, but the significand is wrong.

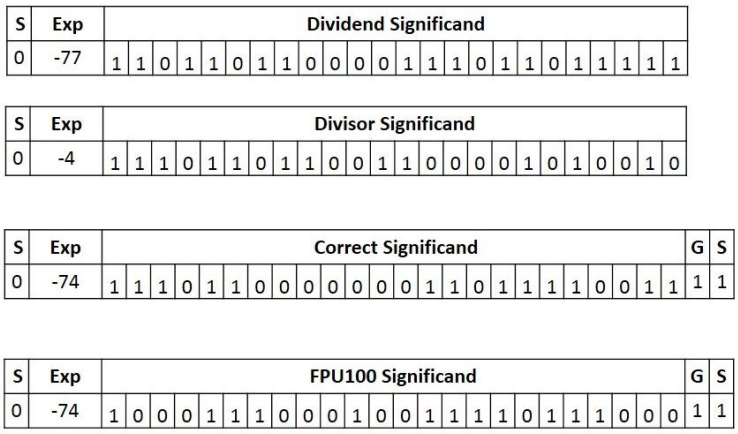

Figure 4.13 Wrong result significand with division when the divisor is greater than the dividend

4.3.1.7. Wrong shifted left version of the result significand in division

The test vector shown in Figure (4.14) shows a shifted version of the result for FPU100 design; the FPU100 result is a shift left version of the correct result by 14.

S	Exp	Dividend Significand
0	-9	1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 0 1 1 1 1 0 1 0 0

S	Exp	Divisor Significand
0	-67	1 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 0 1 1 1 0 0 0

S	Exp	Correct Significand	G	S
0	58	1 0 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 1 1 1 0 0 1 1	0	1

S	Exp	FPU100 Significand	G	S
0	58	1 0 0 1 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 1 1 1 0 1 0	0	1

Figure 4.14 Wrong shifted left version of the result significand in division

4.3.1.8. Wrong result significand and underflow flag, when division result in subnormal number

When dividend and divisor are two normal number and division result in subnormal number that is inexact, the underflow flag should be raised, yet with FPU100, the underflow flag is not raised and the significand value is incorrect as shown in Figure (4.15).

S	Exp	Dividend Significand
0	-52	1 0 0 1 0 0 1 1 1 1 0 1 0 1 0 1 1 0 1 1 1 1 1 1

S	Exp	Divisor Significand
1	74	1 1 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 0 1 1 0 1 0

S	Exp	Correct Significand	G	S
1	-126	0 1 0 1 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 1 1 1	0	1

S	Exp	FPU100 Significand	G	S
1	-126	1 1 0 1 0 0 1 1 0 0 0 0 0 1 0 1 0 1 1 1 1 1 1 0	0	1

Figure 4.15 Wrong result significand and underflow flag, when division result in subnormal number

4.3.1.9. Wrong result and overflow flag when division result in overflow

When dividend exponent minus divisor exponent is 128, which is bigger than $emax$ in single precision, overflow flag should be raised and the result should be infinity, this is not the case with FPU100 as shown in Figure (4.16).

S	Exp	Dividend Significand
1	119	1 1 1 0 1 1 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1 0 0 1

S	Exp	Divisor Significand
1	-9	1 0 1 1 0 1 0 1 0 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1

S	Exp	Correct Significand	G	S
0	Ones	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0	1

S	Exp	FPU100 Significand	G	S
0	127	1 0 0 0 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1	0	1

Figure 4.16 Wrong result and overflow flag when division result in overflow

4.3.1.10. Wrong significand calculation for square root operation

FPU100 design calculates the significand wrongly, Figure (4.17) explains an example for the wrongly generated significand with respect to the correct expected one. The operand significand is equivalent to "1.70644962787628173828125" in decimal, and the square root of such number is "1.3063114", our generated significand is "1.30631137" which is the same as the expected output, yet the FPU100 output is "1.14339876174926757812 5" which is clearly wrong.

S	Exp	Radicand Significand
0	110	1 1 0 1 1 0 1 0 0 1 1 0 1 1 0 0 1 1 1 1 0 0 0 1

S	Exp	Correct Significand
0	55	1 0 1 0 0 1 1 1 0 0 1 1 0 1 0 1 0 0 1 1 0 1 1 0

S	Exp	FPU100 Significand
0	55	1 0 0 1 0 0 1 0 0 1 0 1 1 0 1 0 1 1 1 0 0 1 0 0

Figure 4.17 Wrong significand calculation for square root in FPU100

4.3.2. SYMPL-FP324-AXI4-GP-GPU design

SYMPL FP32X-AXI4 is an open-source, single-precision, multi-thread, IEEE754-2008 compliant, GP-GPU-Compute engine for single or multi-processing floating-point accelerator application written in Verilog RTL [45]. We used our SV constraint model to test the design and we discovered two unique bugs.

4.3.2.1. Wrong left shifted significand value when underflow occurs

When underflow occurs as a result of multiplication operation due to having exponent result less than emin (-126), the result significand is wrong; it is left shifted one bit as shown in Figure (4.18).

S	Exp	Operand1 Significand
1	-12	1.01111000010110100001110

S	Exp	Operand2 Significand
1	-115	1.00110011010100000001010

S	Exp	Correct Significand
0	-127	0.11100001111001001110010

S	Exp	Design Significand
0	-127	0.11000011110010011100100

Figure 4.18 wrong shifted left significand when underflow in FP32X-AXI4

4.3.2.2. Wrong rounding when guard is unset and sticky is set in multiplication

When having a multiplication operation, if the intermediate unbounded result is inexact, where the Guard bit is 0, while the Sticky bits are not, then in rounding, the intermediate result should be corrected by adding one in case of rounding to negative infinity and the sign of the result is negative. This was not handled correctly in FP32X-AXI4 design, where no one was added while rounding as shown in Figure (4.19).

S	Exp	Operand1 Significand
0	-118	1.11010101101000111001111

Rnd to -∞

S	Exp	Operand2 Significand
1	80	1.11000111111010011101000

S	Exp	Intermediate Significand	G	S
1	-36	1.11001111111011111011001	0	1

S	Exp	Correct Significand	G	S
1	-36	1.11001111111011111011010	0	1

S	Exp	Design Significand	G	S
1	-37	1.11001111111011111011001	0	1

Figure 4.19 wrong rounding when having sticky set in FP32X-AXI4 design

4.3.3. Bugs in FPAdd design

FPAdd is one of the FP arithmetic simulators developed in the education of computer arithmetic course for University of Massachusetts Amherst [15]. Following are three bugs discovered by applying our generated test vectors:

4.3.3.1. Wrong Guard value for intermediate result cause wrong value after rounding:

When subtracting two FP numbers and the intermediate result is having one leading zero and guard bit is 1 whilst both round and sticky are 0, doing a left shift will clear the guard bit, so the result is expected to be exact, therefore, rounding stage should not change the result, FPAdd produces wrong rounded results due to having 1 in the guard bit as shown in Figure (4.20).

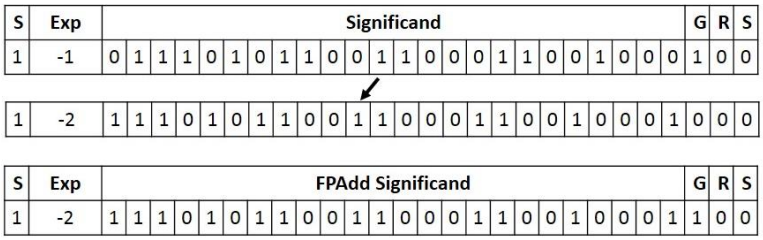

Figure 4.20 Wrong Guard value for intermediate result cause wrong value after rounding

4.3.3.2. Subtracting positive zero from negative zero:

For the following: "(-0)-(+0) =?" the result should be negative zero (-0), yet the site returns positive zero (+0).

4.3.3.3. Wrong inexact and rounding when having a carry with addition

When having a carry in the intermediate result, significand is shifted to the right by 1, and if before shifting, the right most bit of the significand was 1, and the second right most was 0, then after shifting the right most bit is 0, and the guard has 1, if the result is negative and round it to negative infinity, the result significand should be incremented by 1 in rounding step and the result is inexact as shown in Figure (4.21). FPAdd result in exact operation and missed incrementing the significand.

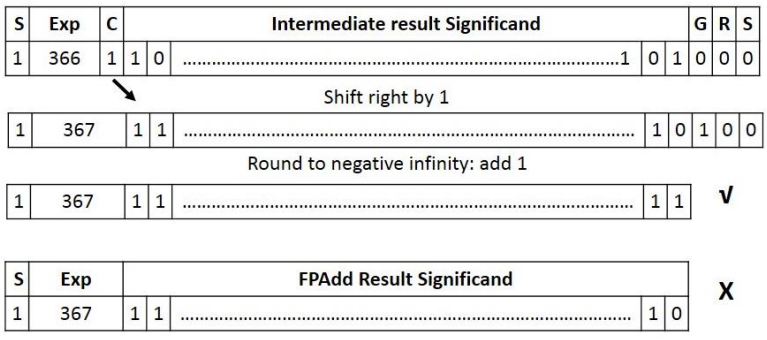

Figure 4.21 Wrong inexact and rounding when having a carry with addition

4.3.4. Double Precision Floating Point Core design (DOUBLE_FPU)

DOUBLE_FPU is an Open cores Verilog design module, IEEE 754-compliant, double precision [41]. 4 operations (addition, subtraction, multiplication, division) are supported, as are the 4 rounding modes (to nearest even, towards zero, to positive and negative infinities). This unit also supports subnormal numbers. Our framework discovered three bugs in addition/subtraction, one in multiplication, and another in division, following are the bugs explained:

4.3.4.1. Wrong implementation of underflow flag in multiplication and division operations

Figures (4.22, 4.23) show a multiplication and a division operation respectively. The one thing common in both test vectors produce underflow due to exponent being less than the minimum possible value. For multiplication, adding both exponents which are (-557) and (-466) result in (-1023) exponent value which is less than emin for double precision (-1022), so underflow flag should be raised, yet with DOUBLE_FPU design, no underflow flag is raised. For division subtracting the divisor exponent (1018) from the dividend exponent (-4) will result in (-1022) which is the minimum exponent, so the result is subnormal, yet the Sticky bits are non-zero, so this should produce underflow flag. DOUBLE_FPU design didn't raise this flag.

S	Exp	Operand 1 Significand
1	-557	1000010011001001010101111100110001001000110100010100 1

S	Exp	Operand 2 Significand
1	-466	1010011001000110110001111010100111110000111101000011 0

S	Exp	Correct Significand (Subnormal)	G	S
0	-1023	0101011000111111010001011000110100100010001110011100 1	0	1

Figure 4.22 Wrong implementation of underflow flag in multiplication

S	Exp	Dividend Significand
1	-4	1000010011001001010101111100110001001000110100010100 1

S	Exp	Divisor Significand
1	1018	1010011001000110110001111010100111110000111101000011 0

S	Exp	Correct Significand (Subnormal)	G	S
0	-1022	0110011000111000001001100100011100110100010010010110 0	0	1

Figure 4.23 Wrong implementation of underflow flag in division operations

4.3.4.2. Wrong Result and inexact flag after rounding due to having non zero sticky bit with addition

When the intermediate result before rounding have zeroes in guard and round but has non zero sticky bits, it is expected that the result is inexact, and if the result is positive and the rounding direction is towards positive infinity, a +1 round value should be added to the rounded significand as shown in the correct significand in Figure (4.24). DOUBLE_FPU rounded significand didn't account for the sticky bits and resulted in wrong result and no raise of inexact flag.

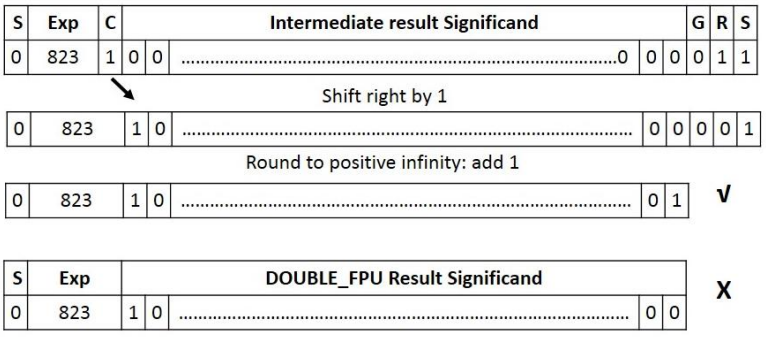

Figure 4.24 Wrong Result and inexact flag after rounding due to having non-zero sticky bit with addition

4.3.4.3. Wrong Result and inexact flag due to skipping sticky bits after the lower operand is normalized

When normalizing the lower operand and we have zeroes in guard and round but has non zero sticky bits, subtracting will result in ones in guard, round and sticky bits, and it

is expected that the result is inexact, and if the result is positive and the rounding direction is towards negative infinity, round value is 0 and nothing should be added in the rounding step as shown in the correct significand in Figure (4.25). DOUBLE_FPU produced wrong result and no raise of inexact flag.

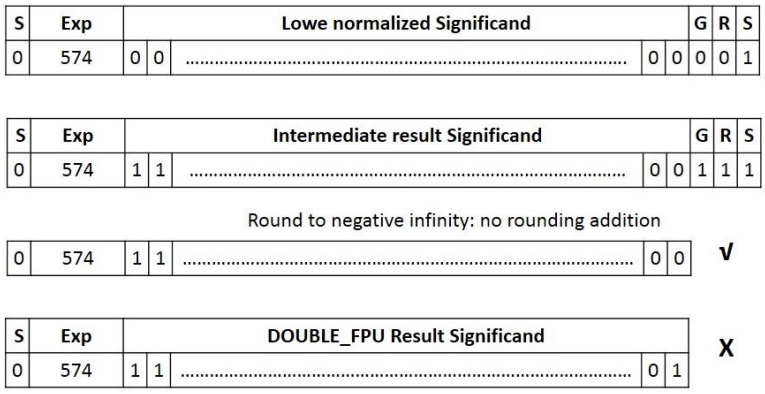

Figure 4.25 Wrong Result and inexact flag due to skipping sticky bits after the lower operand is normalized

4.3.4.4. Wrong rounding when having a carry and round tie even direction with addition

When having a carry in the intermediate result, significand is shifted to the right by 1, and if before shifting, the right most bit of the significand was 1, and the second right most was 0, then after shifting the right most bit is 0, and the guard has 1, whilst round and sticky are 0's, then there is a tie, and hence if the significand is odd, we add 1, else the significand should remain the same, yet, with DOUBLE_FPU, it added 1 making the significand odd as shown in Figure (4.26).

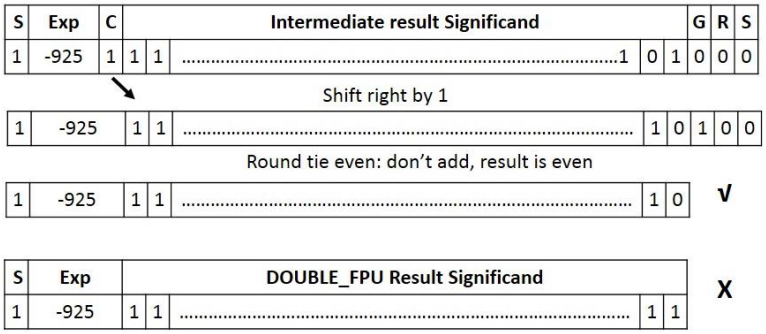

Figure 4.26 wrong rounding when having a carry and round tie even direction with addition

Chapter 5 CONCLUSION AND FUTURE WORK

Verification of floating point unit is a difficult task due to wide inputs that make it impossible to cover all variations. There have not been a generic solution to cover verification of both software and hardware implementations. The implementation of the floating point unit itself impose more complexity in verification due to having long data paths between inputs and outputs. There have never been a generic implementation for floating point units. Also, some operations have iteration approaches involving huge number of cycles to execute and usually FP unit is pipelined. The root cause of floating point bugs is usually due to intermediate result being wrongly rounded, so we need intermediate results to be an input to our verification environment.

Formal and simulation techniques whether simulation or formal have addressed floating point verification. Formal methods prove to be effective, yet they may not be applicable due to failure to create formal model, and are not easily adapted and automated. Also, they involve user interaction. On the other hand, simulation methods are easily implemented yet they cannot guarantee a bug free design.

We proposed a new verification approach based on creating an SV model for the IEEE specification for floating point arithmetic operations namely addition, subtraction, multiplication and division. We used current simulators that supports SV constraints to solve these constraints to generate test vectors for every operation. User defined constraints can be added to be simulated with the SV constraint model to hit corner cases and can be added to intermediate results.

Our verification methodology is simple, generic, time saving and compatible with any simulator that supports SV constraints. The framework shows effectiveness in discovering bugs even for Binary FP addition, subtraction, multiplication and division that has been thoroughly tested since ages.

5.1. Future work

5.1.1. Support more floating point operations

Extending our approach to cover more Binary FP arithmetic operation like namely Fused multiply add and logarithmic functions and applying our generated test vectors on designs supporting these operations will be our first goal.

5.1.2. Support quadruple precision floating point formats

Scaling the verification environment to support quadruple precision will be a good research point since it will increase the random variable count intensively that can reach the limits of simulation tools.

5.1.3. Support Decimal floating point arithmetic

Our work can be extended to support Decimal floating point format and arithmetic. The encoding/decoding constraints should be changed to support the decimal IEEE-754 format. Also, we should add some constraints to handle cohorts.

5.1.4. Extending UVM to use our SV constraints

The Universal Verification Methodology (UVM) is a standardized methodology for verifying integrated circuit designs. UVM is derived mainly from the OVM (Open Verification Methodology). The UVM class library brings much automation to the System Verilog language such as sequences and data automation features (packing, copy, compare) etc. Since based on System Verilog, the library classes can be extended to include our SV constraints library to generate sequences for verification of floating point units.

References

1. "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2008, August 2008.
2. M. Aharoni, S. Asaf, L. Fournier, A. Koifman, R. Nagel, "FPgen - a test generation framework for datapath floating-point verification," High-Level Design Validation and Test Workshop, Eighth IEEE International, pp. 17-22, November 2003.
3. E. Guralnik, M. Aharoni, A.J. Birnbaum, A. Koyfman, "Simulation-Based Verification of Floating-Point Division," Computers, IEEE Transactions, vol.60, no.2, pp.176-188, February 2011.
4. G. Pachiana, J.A. Rodriguez, "Coverage modeling for verification of floating point arithmetic units," Micro-Nanoelectronics, Technology and Applications (EAMTA), Argentine Conference, pp.83-88, July 2014.
5. K. Kannappan, G.H. Herbeck, C. Stearns, "A methodology for automated behavioral verification of floating-point designs," ASIC Conference and Exhibit, Proceedings of Fifth Annual IEEE International, pp.487-490, September 1992.
6. J. Pan, K.N. Levitt, "A Formal Specification of the IEEE Floating-Point Standard with Application to the Verification of F," Signals, Systems and Computers, Conference Record Twenty-Fourth Asilomar Conference, vol.1, pp.505, November 1990.
7. S. Boldo, J.C. Filliatre, "Formal Verification of Floating-Point Programs," Computer Arithmetic, ARITH '07 18th IEEE Symposium, pp.187-194, June 2007.
8. R. Kaivola, N. Narasimhan, "Formal verification of the Pentium(R) 4 multiplier," High-Level Design Validation and Test Workshop, Sixth IEEE International, pp.115-120, 2001.
9. D.W. Matula, L.D. McFearin, "A Formal Model and Efficient Traversal Algorithm for Generating Testbenches for Verification of IEEE Standard Floating Point Division," in Design, Automation and Test in Europe, DATE '06. Proceedings , vol.1, no., pp.1-5, March 2006.
10. M. Aharoni, M. S. Asaf, R. Maharik, I. Nehama, I. Nikulshin, A. Ziv, "Solving constraints on the invisible bits of the intermediate result for floating-point verification," Computer Arithmetic, ARITH '17 17th IEEE Symposium, pp.76-83, June 2005.
11. A.A. Sayed-Ahmed, H.A. Fahmy, M.Y. Hassan, "Three engines to solve verification constraints of decimal Floating-Point operation," Signals, Systems and Computers (ASILOMAR), Conference Record of the Forty Fourth Asilomar Conference, pp.1153-1157, November 2010.
12. A.A. Sayed-Ahmed, H.A. Fahmy, R. Samy, "Verification of decimal floating-point fused-multiply-add operation," Computer Systems and Applications (AICCSA), 9th IEEE/ACS International Conference, pp.255-262, December 2011.
13. OpenCores, "FPU100: overview," captured in: http://www. opencores.org/?do=project&who=fpu100, August 2009.

14. “IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2012, February 2013.
15. Umass, “Floating Point Add/Subtract,” captured in: http://www.ecs.umass.edu./ece/koren/arith/simulator/FPAdd/, December 2012.
16. M. Schoeberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems", Vienna University of Technology, PhD Thesis, January 2005.
17. D. Price, "Pentium FDIV flaw-lessons learned," Micro, IEEE, vol.15, no.2, pp.86,88, April 1995.
18. Jean-Christophe Filliâtre, Claude March´e and Thierry Hubert. The Caduceus tool for the verification of C programs. http://caduceus.lri.fr/.
19. The Coq Proof Assistant. http://coq.inria.fr/.
20. R. E. Bryant., “On the complexity of VLSI implementations and graph representations of boolean functions with applications to integer multiplication,” Computers, IEEE Transaction, vol. 40, no. 2, pp. 205–213, February 1991.
21. D. M. Russinoff, “A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor,” London Mathematical Society Journal of Computational Mathematics, pp.148–200, 1998.
22. V.D. Thai, T.T. Quan, T.V. Le, B.T. Ngo, "An Approximation-Based Abstract Interpretation Framework for Formal Verification of Floating-Point Programs," in Computing and Communication Technologies, IEEE RIVF International Conference, pp.1-4, February 2012.
23. M.D. Aagaard, C.-J.H. Seger, "The formal verification of a pipelined double-precision IEEE floating-point multiplier," in Computer-Aided Design, IEEE/ACM International Conference, pp.7-10, November 1995.
24. N. Kikkeri, P.-M. Seidel, "Optimized Arithmetic Hardware Design based on Hierarchical Formal Verification," in Electronics, Circuits and Systems, ICECS '06 13th IEEE International Conference, pp.541-544, December 2006.
25. P-M. Seidel,; G. Even, “Delay-optimized implementation of IEEE floating-point addition,” IEEE Transactions on Computers, no.2, vol.53, pp.97-114, February 2004.
26. G. Even, P-M. Seidel, ”A comparison of three rounding algorithms for IEEE floating-point multiplication,” IEEE Transactions on Computers, no. 7, vol.49, pp. 638-650, July 2000.
27. M. Brain, C. Tinelli, P. Ruemmer, T. Wahl, "An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic," in Computer Arithmetic, IEEE 22nd Symposium, pp.160-167, June 2015.
28. U. Krautz, V. Paruthi, A. Arunagiri, S. Kumar, S. Pujar, T. Babinsky, "Automatic verification of Floating Point Units," in Design Automation Conference (DAC), 51st ACM/EDAC/IEEE, pp.1-6, June 2014.
29. J. Rupley, J. King, E. Quinnell, F. Galloway, K. Patton, P. Seidel, J. Dinh, H. Bui; A. Bhowmik, "The Floating-Point Unit of the Jaguar x86 Core," in Computer Arithmetic (ARITH), 21st IEEE Symposium, pp.7-16, April 2013.

30. V.M. KiranKumar, A. Gupta, R. Ghughal, "Symbolic Trajectory Evaluation: The primary validation Vehicle for next generation Intel® Processor Graphics FPU," in Formal Methods in Computer-Aided Design (FMCAD), pp.149-156, October 2012.

31. O. Goni, E. Todorovich, "Components for Coverage-Driven Verification of floating-point units," in Programmable Logic (SPL), IX Southern Conference, pp.1-7, November 2014.

32. M. Glasser, Open Verification Methodology Cookbook. Springer Dordrecht Heidelberg London New York, 2009.

33. O. Goni, M. Vazquez, E. Todorovich, G. Sutter, "Experiences applying framework-based functional verification to a design for programmable logic," in Programmable Logic (SPL), VII Southern Conference, pp.109-115, April 2011.

34. M. Cowlishaw, The decNumber C library, 3rd ed., IBM Corporation, 2008.

35. Y-A. Chen; R.E. Bryant, "*PHDD: an efficient graph representation for floating point circuit verification," in Computer-Aided Design, IEEE/ACM International Conference, pp.2-7, November 1997.

36. Q. Wang, X. Song; W.N. Hung, M. Gu, J. Sun, "Scalable Verification of a Generic End-Around-Carry Adder for Floating-Point Units by Coq," in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions, vol.34, no.1, pp.150-154, January 2015.

37. C-I. Chen, C-Y. Yu, Y-J. Lu, C-F. Wu, "Apply high-level synthesis design and verification methodology on floating-point unit implementation," in VLSI Design, Automation and Test (VLSI-DAT), International Symposium, pp.1-4, April 2014.

38. D.W. Matula, L.D. McFearin, "A Formal Model and Efficient Traversal Algorithm for Generating Testbenches for Verification of IEEE Standard Floating Point Division," in Design, Automation and Test in Europe, DATE '06. Proceedings , vol.1, no., pp.1-5, March 2006.

39. A. Ziv, M. Aharoni, S. Asaf, "Solving range constraints for binary floating-point instructions," in Computer Arithmetic, 16th IEEE Symposium, pp.158-164, June 2003.

40. E. Guralnik, A.J. Birnbaum, A. Koyfman, A. Kaplan, "Implementation Specific Verification of Divide and Square Root Instructions," in Computer Arithmetic, 19th IEEE Symposium, pp.114-121, June 2009.

41. M. Aharoni, R. Maharik, A. Ziv, "Solving Constraints on the Intermediate Result of Decimal Floating-Point Operations," in Computer Arithmetic, ARITH '07. 18th IEEE Symposium on, pp.38-45, June 2007.

42. D. Lundgren, "Double Precision Floating Point Core Verilog," captured in: http://opencores.org/project,double_fpu, December 2009.

43. R.V.W. Putra, "A novel fixed-point square root algorithm and its digital hardware design," in ICT for Smart Society (ICISS), pp.1-4, June 2013.

44. A.S. Ahmed, H. Fahmy, U. Kuhne, "Verification of the decimal floating-point square root operation," in Test Symposium (ETS), 19th IEEE European , pp.1-2, May 2014.

45. J. Harthcock, "Open-source, single-precision, GP-GPU-Compute engine being designed for IEEE754 compliance," captured in: https://github.com/jerry-D/SYMPL-FP324-AXI4-GP-GPU, September 2015.

Appendix A: SV constraints for Addition/Subtraction

```
`define N 1000  //Number of generated test vectors
`define k 32 //Change to 64 for double
`define p 24 //Change to 53 for double
`define w 8 //Change to 11 for double
`define emax 127 //Change to 1023 for double
`define bias 127 //Change to 1023 for double
module DUT_normal;
 class floating_point_numbers_variables;
  typedef enum {ADD,SUB} operationTypes;
  rand operationTypes operation,effectiveOperation;
  rand bit [`k-1:0] operand1,operand2,result;
 typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
  rand roundTypes roundDirection;
  rand bit operand1Sign,operand2Sign,
  resultSign,higherSign,lowerSign,
  intermediateSign,intermediateNormalizedSign,
  roundSign,roundNormalizedSign;
  rand bit [`w-1:0] operand1Exponent,operand2Exponent,
  resultExponent,higherExponent,lowerExponent,
  intermediateExponent,intermediateNormalizedExponent,
  roundExponent,roundNormalizedExponent;
  rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
  rand bit [`p-1:0] operand1Significand,operand2Significand,
  resultSignificand,higherSignificand,lowerSignificand,lowerNormalizedSignificand,
  intermediateSignificand,intermediateNormalizedSignificand,
  roundSignificand,roundNormalizedSignificand;
  rand bit carry,roundCarry,roundValue;
  rand bit lowerGuard,lowerRound,lowerSticky;
  rand bit intermediateGuard,intermediateRound,intermediateSticky;
  rand bit intermediateNormalizedGuard,intermediateNormalizedRound,
  intermediateNormalizedSticky;
  rand int intermediateShiftValue,intermediateNormalizedShiftValue;
  rand bit inexactFlag,overflowFlag;
  const bit invalidFlag = 1'b0;
  const bit underflowFlag = 1'b0;
  const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
  isOperand1Inf = 1'b0,isOperand2Inf = 1'b0;
  const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b0,
  isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;
  constraint binary_encoding_decoding {
  {operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
  {operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
  {resultSign,resultExponent,resultMantissa} == result;
  }
  constraint is_operands_infinity_NaN_zero_subnormal {
    if (isOperand1NaN)
```

```
  ((operand1Exponent == '1) && (operand1Mantissa != '0));
 else if (isOperand1Inf)
  ((operand1Exponent == '1) && (operand1Mantissa == '0));
 else
  operand1Exponent != '1 && operand1Exponent != '0;
 if (isOperand2NaN)
  ((operand2Exponent == '1) && (operand2Mantissa != '0));
 else if (isOperand2Inf)
  ((operand2Exponent == '1) && (operand2Mantissa == '0));
 else
  operand2Exponent != '1 && operand2Exponent != '0;
}
constraint significand_mantissa {
  ({1'b1,operand1Mantissa} == operand1Significand);
  ({1'b1,operand2Mantissa} == operand2Significand);
}
constraint higher_lower_exponent {
  if (operand1Exponent > operand2Exponent)  (
  (higherSign == operand1Sign) && (higherExponent == operand1Exponent) &&
  (higherSignificand == operand1Significand) && (lowerSign == operand2Sign) &&
  (lowerExponent == operand2Exponent) &&
  (lowerSignificand == operand2Significand) );
  else if (operand1Exponent < operand2Exponent) (
  (higherSign == operand2Sign) && (higherExponent == operand2Exponent) &&
  (higherSignificand == operand2Significand) && (lowerSign == operand1Sign) &&
  (lowerExponent == operand1Exponent) &&
  (lowerSignificand == operand1Significand) );
  else if (operand1Significand >= operand2Significand) (
  (higherSign == operand1Sign) && (higherExponent == operand1Exponent) &&
  (higherSignificand == operand1Significand) && (lowerSign == operand2Sign) &&
  (lowerExponent == operand2Exponent) &&
  (lowerSignificand == operand2Significand) );
  else (
  (higherSign == operand2Sign) && (higherExponent == operand2Exponent) &&
  (higherSignificand == operand2Significand) && (lowerSign == operand1Sign) &&
  (lowerExponent == operand1Exponent) &&
  (lowerSignificand == operand1Significand) );
}
constraint lower_normalized {
 {lowerNormalizedSignificand,lowerGuard,lowerRound} ==
 {lowerSignificand,2'b0} >> (higherExponent-lowerExponent);
  if (higherExponent-lowerExponent < `p+2)
    lowerSticky ==
    (|(lowerSignificand << ((`p+2)-(higherExponent-lowerExponent))));
  else
    lowerSticky == (|lowerSignificand);
 }
 constraint effective_operation {
  if ((operation == ADD) && (higherSign == lowerSign))
  (effectiveOperation == ADD);
```

```
  else if ((operation == ADD) && (higherSign != lowerSign))
  (effectiveOperation == SUB);
  else if         ((operation == SUB) && (higherSign == lowerSign))
  (effectiveOperation == SUB);
  else (effectiveOperation == ADD);
}
constraint addition_subtraction {
  if (effectiveOperation == ADD)
  {carry,intermediateSignificand,
   intermediateGuard,intermediateRound,intermediateSticky} ==
  {1'b0,lowerNormalizedSignificand,lowerGuard,lowerRound,lowerSticky} +
  {1'b0,higherSignificand,3'b0};
  else
  {carry,intermediateSignificand,
  intermediateGuard,intermediateRound,intermediateSticky} ==
  {1'b0,higherSignificand,3'b0} –
  {1'b0,lowerNormalizedSignificand,lowerGuard,lowerRound,lowerSticky};
  if ((effectiveOperation == ADD) && (operation == ADD))
  (intermediateSign == higherSign);
  else if         ((effectiveOperation == ADD) && (operation == SUB))
  (intermediateSign == operand1Sign);
  else if         ((effectiveOperation == SUB) && (operation == ADD))
  (intermediateSign == higherSign);
  else (intermediateSign ==
  (((operand1Exponent < operand2Exponent) ||
  ((operand1Exponent == operand2Exponent) &&
  (operand1Significand < operand2Significand))) ^(operand1Sign)));
  intermediateExponent == higherExponent;
}
function int leading_zero_calculation (input [0:`p+2] functionSignificand);
  for (int i = 0; i <= `p+2;i++)
    if (functionSignificand[i] == 1'b1) return i;
    return `p+3;
endfunction
constraint carry_leading_zero_correction {
  intermediateSign == intermediateNormalizedSign;
  intermediateShiftValue == leading_zero_calculation(intermediateSignificand);
  if (intermediateShiftValue >= intermediateExponent)
    intermediateNormalizedShiftValue == intermediateExponent-1;
  else intermediateNormalizedShiftValue == intermediateShiftValue;
  // only valid in addition
  if (carry == 1'b1) (
    (intermediateNormalizedExponent == intermediateExponent + 1'b1) &&
    (intermediateNormalizedGuard == intermediateSignificand[0]) &&
    (intermediateNormalizedRound == intermediateGuard) &&
    (intermediateNormalizedSticky == (intermediateSticky | intermediateRound)) &&
    (intermediateNormalizedSignificand == {1'b1,intermediateSignificand[`p-1:1]}));
  // no leading zeros
  else if((intermediateSignificand[`p-1] == 1'b1)) (
    (intermediateNormalizedExponent == intermediateExponent ) &&
```

```
    (intermediateNormalizedGuard == intermediateGuard) &&
    (intermediateNormalizedRound == intermediateRound) &&
    (intermediateNormalizedSticky == intermediateSticky) &&
    (intermediateNormalizedSignificand == intermediateSignificand));
   else if ((intermediateShiftValue >= intermediateExponent)) (
    (intermediateNormalizedExponent == '0) &&
    (intermediateNormalizedGuard == 1'b0) &&
    (intermediateNormalizedRound == 1'b0) &&
    (intermediateNormalizedSticky == 1'b0) &&
    ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
    intermediateNormalizedRound,intermediateNormalizedSticky} ==
    ({intermediateSignificand,intermediateGuard,
    intermediateRound,intermediateSticky} << intermediateNormalizedShiftValue)));
   else if ((intermediateShiftValue != `p+3)) (
    (intermediateNormalizedExponent ==
    intermediateExponent - intermediateNormalizedShiftValue) &&
    (intermediateNormalizedGuard == 1'b0) &&
    (intermediateNormalizedRound == 1'b0) &&
    (intermediateNormalizedSticky == 1'b0) &&
    ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
    intermediateNormalizedRound,intermediateNormalizedSticky} ==
    ({intermediateSignificand,intermediateGuard,
    intermediateRound,intermediateSticky} << intermediateNormalizedShiftValue)));
   else (
   (intermediateNormalizedExponent == '0) &&
   (intermediateNormalizedGuard == 1'b0) &&
   (intermediateNormalizedRound == 1'b0) &&
   (intermediateNormalizedSticky == 1'b0) &&
   ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
   intermediateNormalizedRound,intermediateNormalizedSticky} == '0));
  }
  constraint rounding {
   (roundDirection == roundZero) -> (roundValue == 1'b0);
   (roundDirection == roundPositive) ->
   (roundValue == (~intermediateNormalizedSign &
   (intermediateNormalizedGuard | intermediateNormalizedRound |
   intermediateNormalizedSticky)));
   (roundDirection == roundNegative) ->
   (roundValue ==  (intermediateNormalizedSign & (intermediateNormalizedGuard |
   intermediateNormalizedRound | intermediateNormalizedSticky)));
   (roundDirection == roundTieEven) ->
   (roundValue == (intermediateNormalizedGuard & (intermediateNormalizedRound
   | intermediateNormalizedSticky | intermediateNormalizedSignificand[0])));
  }
  constraint addition_after_round {
   {roundCarry,roundSignificand} == ({1'b0,intermediateNormalizedSignificand}
   + roundValue);
   roundSign == intermediateNormalizedSign;
   roundExponent == intermediateNormalizedExponent;
   }
```

```
  constraint normalization_after_rounding {
   roundNormalizedSign == roundSign;
   (roundCarry == 1'b1) -> (
   (roundNormalizedExponent == roundExponent + 1) &&
   (roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
   (roundCarry == 1'b0) -> (
   (roundNormalizedExponent == roundExponent) &&
   (roundNormalizedSignificand == roundSignificand));
  }
  constraint overflow_flag {
   overflowFlag == ((roundNormalizedExponent == '1) |
   (intermediateNormalizedExponent == '1));
  }
  constraint inexact_flag {
   inexactFlag == (intermediateNormalizedGuard | intermediateNormalizedRound |
   intermediateNormalizedSticky | overflowFlag);
  }
  constraint result_calculation {
   if ((overflowFlag == 1'b0)) (
     (resultSign == roundNormalizedSign) &&
     (resultExponent == roundNormalizedExponent) &&
     (resultSignificand == roundNormalizedSignificand) &&
     (resultMantissa == roundNormalizedSignificand[`p-2:0])
   );
   else if (roundDirection == roundTieEven) (
     (resultSign == roundNormalizedSign) &&
     (resultExponent == '1) &&
     (resultSignificand == {1'b1,'0}) &&
     (resultMantissa == '0)
   );
   else if ((roundDirection == roundZero) ||
   ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b1)) ||
   ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b0))) (
     (resultSign == roundNormalizedSign) &&
     (resultExponent == `bias+`bias) &&
     (resultSignificand == '1) &&
     (resultMantissa == '1)
   );
    else if ((roundDirection == roundTieEven) ||
    ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b0)) ||
    ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b1))) (
      (resultSign == roundNormalizedSign) &&
      (resultExponent == '1) &&
      (resultSignificand == {1'b1,'0}) &&
      (resultMantissa == '0)
    );
  }
endclass
int i;
floating_point_numbers_variables fpv;
```

```
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);
$display("Flags : %b %b %b
%b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
module DUT_subnormal;
class floating_point_numbers_variables;
  typedef enum {ADD,SUB} operationTypes;
  rand operationTypes operation,effectiveOperation;
  rand bit [`k-1:0] operand1,operand2,result;
  typedef enum {roundTieEven,roundPositive,roundNegative,roundZero}
  roundTypes;
  rand roundTypes roundDirection;
  rand bit operand1Sign,operand2Sign,resultSign,
  higherSign,lowerSign,intermediateSign,intermediateNormalizedSign,
  roundSign,roundNormalizedSign;
  rand bit [`w-1:0] operand1Exponent,resultExponent,higherExponent,
  intermediateExponent,intermediateNormalizedExponent,
  roundExponent,roundNormalizedExponent;
  const bit [`w-1:0] operand2Exponent = '0,lowerExponent = '0;
  rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
  rand bit [`p-1:0] operand1Significand,operand2Significand,resultSignificand,
  higherSignificand,lowerSignificand,lowerNormalizedSignificand,
  intermediateSignificand,intermediateNormalizedSignificand,
  roundSignificand,roundNormalizedSignificand;
  rand bit carry,roundCarry,roundValue;
  rand bit lowerGuard,lowerRound,lowerSticky;
  rand bit intermediateGuard,intermediateRound,intermediateSticky;
  rand bit intermediateNormalizedGuard,intermediateNormalizedRound,
  intermediateNormalizedSticky;
  rand int intermediateShiftValue,intermediateNormalizedShiftValue;
  rand bit inexactFlag,overflowFlag;
  const bit invalidFlag = 1'b0;;
  const bit underflowFlag = 1'b0;
  const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
  isOperand1Inf =   1'b0,isOperand2Inf = 1'b0;
  const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b0,
  isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;;
  constraint binary_encoding_decoding {
    {operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
    {operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
```

```
  {resultSign,resultExponent,resultMantissa} == result;
}
constraint is_operands_infinity_NaN_zero_subnormal {
  operand1Exponent != '1 && operand1Exponent != '0;
  operand2Exponent == '0;
}
constraint significand_mantissa {
  ({1'b1,operand1Mantissa} == operand1Significand) ;
  ({1'b0,operand2Mantissa} == operand2Significand) ;
}
constraint higher_lower_exponent {
  (higherSign == operand1Sign) && (higherExponent == operand1Exponent) &&
  (higherSignificand == operand1Significand);
  (lowerSign == operand2Sign) && (lowerSignificand == operand2Significand);
}
constraint lower_normalized {
  {lowerNormalizedSignificand,lowerGuard,lowerRound} ==
  {lowerSignificand,2'b0} >> (higherExponent-1);
  if (higherExponent < `p+2)
    lowerSticky == (|(lowerSignificand << ((`p+2)-(higherExponent-1))));
  else
    lowerSticky == (|lowerSignificand);
}
constraint effective_operation {
  if ((operation == ADD) && (higherSign == lowerSign))
    (effectiveOperation == ADD);
  else if ((operation == ADD) && (higherSign != lowerSign))
    (effectiveOperation == SUB);
  else if          ((operation == SUB) && (higherSign == lowerSign))
    (effectiveOperation == SUB);
  else (effectiveOperation == ADD);
}
constraint addition_subtraction {
  if (effectiveOperation == ADD)
    {carry,intermediateSignificand,
    intermediateGuard,intermediateRound,intermediateSticky} ==
   {1'b0,lowerNormalizedSignificand,lowerGuard,lowerRound,lowerSticky} +
   {1'b0,higherSignificand,3'b0};
 else
  {carry,intermediateSignificand,
  intermediateGuard,intermediateRound,intermediateSticky} ==
  {1'b0,higherSignificand,3'b0} –
  {1'b0,lowerNormalizedSignificand,lowerGuard,lowerRound,lowerSticky};
  intermediateSign == higherSign;
  intermediateExponent == higherExponent;
}
function int leading_zero_calculation (input [0:`p+2] functionSignificand);
  for (int i = 0; i <= `p+2;i++)
    if (functionSignificand[i] == 1'b1) return i;
  return `p+3;
```

```
 endfunction
 constraint carry_leading_zero_correction {
  intermediateSign == intermediateNormalizedSign;
  intermediateShiftValue == leading_zero_calculation(intermediateSignificand);
  if (intermediateShiftValue >= intermediateExponent)
  intermediateNormalizedShiftValue == intermediateExponent-1;
  else intermediateNormalizedShiftValue == intermediateShiftValue;
  if (carry == 1'b1) (
    (intermediateNormalizedExponent == intermediateExponent + 1'b1) &&
    (intermediateNormalizedGuard == intermediateSignificand[0]) &&
    (intermediateNormalizedRound == intermediateGuard) &&
    (intermediateNormalizedSticky == (intermediateSticky | intermediateRound)) &&
    (intermediateNormalizedSignificand == {1'b1,intermediateSignificand[`p-1:1]}));
  else if((intermediateSignificand[`p-1] == 1'b1)) (
    (intermediateNormalizedExponent == intermediateExponent ) &&
    (intermediateNormalizedGuard == intermediateGuard) &&
    (intermediateNormalizedRound == intermediateRound) &&
    (intermediateNormalizedSticky == intermediateSticky) &&
    (intermediateNormalizedSignificand == intermediateSignificand));
  else if ((intermediateShiftValue >= intermediateExponent)) (
   (intermediateNormalizedExponent == '0) &&
   (intermediateNormalizedGuard == 1'b0) &&
   (intermediateNormalizedRound == 1'b0) &&
   (intermediateNormalizedSticky == 1'b0) &&
   ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
   intermediateNormalizedRound,intermediateNormalizedSticky} ==
   ({intermediateSignificand,intermediateGuard,
   intermediateRound,intermediateSticky} << intermediateNormalizedShiftValue)));
  else if ((intermediateShiftValue != `p+3)) (
   (intermediateNormalizedExponent ==
   intermediateExponent - intermediateNormalizedShiftValue) &&
   (intermediateNormalizedGuard == 1'b0) &&
   (intermediateNormalizedRound == 1'b0) &&
   (intermediateNormalizedSticky == 1'b0) &&
   ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
   intermediateNormalizedRound,intermediateNormalizedSticky} ==
   ({intermediateSignificand,intermediateGuard,intermediateRound,
   intermediateSticky} << intermediateNormalizedShiftValue)));
  else (
   (intermediateNormalizedExponent == '0) &&
   (intermediateNormalizedGuard == 1'b0) &&
   (intermediateNormalizedRound == 1'b0) &&
   (intermediateNormalizedSticky == 1'b0) &&
   ({intermediateNormalizedSignificand,intermediateNormalizedGuard,
   intermediateNormalizedRound,intermediateNormalizedSticky} == '0));
 }
 constraint rounding {
   (roundDirection == roundZero) -> (roundValue == 1'b0);
   (roundDirection == roundPositive) ->
   (roundValue == (~intermediateNormalizedSign &
```

```
  (intermediateNormalizedGuard | intermediateNormalizedRound
  | intermediateNormalizedSticky)));
  (roundDirection == roundNegative) ->
  (roundValue == (intermediateNormalizedSign & (intermediateNormalizedGuard |
  intermediateNormalizedRound | intermediateNormalizedSticky)));
  (roundDirection == roundTieEven) ->
  (roundValue == (intermediateNormalizedGuard & (intermediateNormalizedRound
  | intermediateNormalizedSticky | intermediateNormalizedSignificand[0])));
}
constraint addition_after_round {
  {roundCarry,roundSignificand} == ({1'b0,intermediateNormalizedSignificand} +
  roundValue);
  roundSign == intermediateNormalizedSign;
  roundExponent == intermediateNormalizedExponent;
}
constraint normalization_after_rounding {
  roundNormalizedSign == roundSign;
  (roundCarry == 1'b1) -> (
  (roundNormalizedExponent == roundExponent + 1) &&
  (roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
  (roundCarry == 1'b0) -> (
  (roundNormalizedExponent == roundExponent) &&
  (roundNormalizedSignificand == roundSignificand));
}
constraint overflow_flag {
  overflowFlag == ((roundNormalizedExponent == '1) |
  (intermediateNormalizedExponent == '1));
}
constraint inexact_flag {
  inexactFlag == (intermediateNormalizedGuard | intermediateNormalizedRound |
  intermediateNormalizedSticky | overflowFlag);
}
constraint result_calculation {
  if ((overflowFlag == 1'b0)) (
    (resultSign == roundNormalizedSign) &&
    (resultExponent == roundNormalizedExponent) &&
    (resultSignificand == roundNormalizedSignificand) &&
    (resultMantissa == roundNormalizedSignificand[`p-2:0]));
  else if (roundDirection == roundTieEven) (
    (resultSign == roundNormalizedSign) &&
    (resultExponent == '1) &&
    (resultSignificand == {1'b1,'0}) &&
    (resultMantissa == '0));
  else if ((roundDirection == roundZero) ||
  ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b1)) ||
  ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b0))) (
    (resultSign == roundNormalizedSign) &&
    (resultExponent == `bias+`bias) &&
    (resultSignificand == '1) &&
   (resultMantissa == '1));
```

```
    else if ((roundDirection == roundTieEven)
    || ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b0)) ||
    ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b1))) (
      (resultSign == roundNormalizedSign) &&
      (resultExponent == '1) &&
      (resultSignificand == {1'b1,'0}) &&
      (resultMantissa == '0));
  }
endclass
int i;
floating_point_numbers_variables fpv;
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);
$display("Flags                                    :    %b    %b    %b
%b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
```

Appendix B: SV constraints for Multiplication

```
`define N 1000  //Number of generated test vectors
`define k 32 //Change to 64 for double
`define p 24 //Change to 53 for double
`define w 8 //Change to 11 for double
`define emax 127 //Change to 1023 for double
`define bias 127 //Change to 1023 for double
module DUT_normal;
class floating_point_numbers_variables;
rand bit [`k-1:0] operand1,operand2,result;
typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
rand roundTypes roundDirection;
rand bit operand1Sign,operand2Sign,resultSign,
intermediateSign,intermediateNormalizedSign,
roundSign,roundNormalizedSign;
rand bit [`w-1:0] operand1Exponent,operand2Exponent,resultExponent,
operand1NormalizedExponent,operand2NormalizedExponent;
rand bit [`w:0] intermediateExponent,intermediateNormalizedExponent,
roundExponent,roundNormalizedExponent;
rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
rand bit [`p-1:0] operand1Significand,operand2Significand,resultSignificand,
intermediateSignificand,intermediateNormalizedSignificand,
roundSignificand,roundNormalizedSignificand;
rand bit [`p-2:0] intermediateGuardSticky;
rand bit carry,roundCarry,roundValue;
rand bit intermediateGuard,intermediateSticky;
rand bit intermediateNormalizedGuard,intermediateNormalizedSticky;
rand bit inexactFlag,overflowFlag,underflowFlag;
const bit invalidFlag = 1'b0;
const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
isOperand1Inf = 1'b0,isOperand2Inf = 1'b0;
const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b0,
isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;
rand int intermediateNormalizedShiftValue;
constraint binary_encoding_decoding {
{operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
{operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
{resultSign,resultExponent,resultMantissa} == result;
}
constraint is_operands_infinity_NaN_zero_subnormal {
if (isOperand1NaN)
((operand1Exponent == '1) && (operand1Mantissa != '0));
else if (isOperand1Inf)
((operand1Exponent == '1) && (operand1Mantissa == '0));
else
operand1Exponent != '1 && operand1Exponent != '0;
if (isOperand2NaN)
```

```
((operand2Exponent == '1) && (operand2Mantissa != '0));
else if (isOperand2Inf)
((operand2Exponent == '1) && (operand2Mantissa == '0));
else
operand2Exponent != '1 && operand2Exponent != '0;
}
constraint significand_mantissa  {
({1'b1,operand1Mantissa} == operand1Significand) &&
 (operand1NormalizedExponent == operand1Exponent) ;
({1'b1,operand2Mantissa} == operand2Significand) &&
 (operand2NormalizedExponent == operand2Exponent) ;
}
function [2*`p-1:0] partial_product_multiplication(input [`p-1:0] op1,op2);
bit [2*`p-1:0] normal[0:`p-1], shifted[0:`p];
shifted[0] = 0;
for (int i = 0; i <= `p-1; i++)
begin
if (op2[i])
normal[i] = {`p'b0,op1};
else
normal[i] = '0;
shifted[i+1] = (normal[i] << i) + shifted[i];
end
return shifted[`p];
endfunction
constraint multiplication {
{carry,intermediateSignificand,intermediateGuardSticky} ==
 partial_product_multiplication(operand1Significand,operand2Significand);
intermediateGuard == intermediateGuardSticky[`p-2];
intermediateSticky == (|intermediateGuardSticky[`p-3:0]);
intermediateSign == (operand1Sign ^ operand2Sign);
intermediateExponent == (operand1Exponent + operand2Exponent - `bias);
}
constraint underflow_flag {
if (intermediateExponent > 0 || carry) underflowFlag  == 1'b0;
else  underflowFlag == 1'b1;
}
constraint carry_correction {
intermediateSign == intermediateNormalizedSign;
(carry == 1'b1) -> (
(intermediateNormalizedExponent == intermediateExponent + 1'b1) &&
(intermediateNormalizedGuard == intermediateSignificand[0]) &&
(intermediateNormalizedSticky == (intermediateSticky | intermediateGuard)) &&
(intermediateNormalizedSignificand == {1'b1,intermediateSignificand[`p-1:1]}));
(carry == 1'b0 && underflowFlag == 1'b0) -> (
(intermediateNormalizedExponent == (intermediateExponent)) &&
(intermediateNormalizedGuard == intermediateGuard) &&
(intermediateNormalizedSticky == intermediateSticky) &&
(intermediateNormalizedSignificand == intermediateSignificand));
(carry == 1'b0 && underflowFlag == 1'b1) -> (
```

```
(intermediateNormalizedExponent == (intermediateExponent)) &&
(intermediateNormalizedGuard == intermediateSignificand[0]) &&
(intermediateNormalizedSticky == (intermediateSticky | intermediateGuard)) &&
(intermediateNormalizedSignificand == {1'b0,intermediateSignificand[`p-1:1]}));
}
constraint overflow_flag {
if (intermediateNormalizedExponent < (`bias+`bias+1)) overflowFlag == 1'b0;
else overflowFlag == 1'b1;
}
constraint inexact_flag {
inexactFlag == (intermediateNormalizedGuard | intermediateNormalizedSticky |
overflowFlag | underflowFlag);
}
constraint rounding {
(roundDirection == roundZero) -> (roundValue == 1'b0);
(roundDirection == roundPositive) -> (roundValue == (~intermediateNormalizedSign
& (intermediateNormalizedGuard | intermediateNormalizedSticky)));
(roundDirection == roundNegative) -> (roundValue == (intermediateNormalizedSign
& (intermediateNormalizedGuard | intermediateNormalizedSticky)));
(roundDirection == roundTieEven) -> (roundValue ==
(intermediateNormalizedGuard & (intermediateNormalizedSticky |
intermediateNormalizedSignificand[0])));
}
constraint addition_after_round {
{roundCarry,roundSignificand} == ({1'b0,intermediateNormalizedSignificand} +
roundValue);
roundSign == intermediateNormalizedSign;
roundExponent == intermediateNormalizedExponent;
}
constraint normalization_after_rounding {
roundNormalizedSign == roundSign;
(roundCarry == 1'b1) -> (
(roundNormalizedExponent == roundExponent + 1) &&
(roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
(roundCarry == 1'b0) -> (
(roundNormalizedExponent == roundExponent) &&
(roundNormalizedSignificand == roundSignificand));
}
constraint result_calculation {
if (overflowFlag && (roundDirection == roundTieEven))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0)
;
else if (overflowFlag && ((roundDirection == roundZero) || ((roundDirection ==
roundPositive) && (roundNormalizedSign == 1'b1)) || ((roundDirection ==
roundNegative) && (roundNormalizedSign == 1'b0))))
(resultSign == roundNormalizedSign) &&
(resultExponent == `bias+`bias) &&
```

```
(resultSignificand == '1) &&
(resultMantissa == '1)
;
else if (overflowFlag && ((roundDirection == roundTieEven) || ((roundDirection ==
roundPositive) && (roundNormalizedSign == 1'b0)) || ((roundDirection ==
roundNegative) && (roundNormalizedSign == 1'b1))))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0)
;
else
(resultSign == roundNormalizedSign) &&
(resultExponent == roundNormalizedExponent[`w-1:0]) &&
(resultSignificand == roundNormalizedSignificand) &&
(resultMantissa == roundNormalizedSignificand[`p-2:0])
;
}

endclass
int i;
floating_point_numbers_variables fpv;
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);
$display("Flags : %b %b %b
%b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
module DUT_subnormal;
class floating_point_numbers_variables;
rand bit [`k-1:0] operand1,operand2,result;
typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
rand roundTypes roundDirection;
rand bit operand1Sign,operand2Sign,resultSign,
intermediateSign,intermediateNormalizedSign,roundSign,roundNormalizedSign;
rand bit [`w-1:0] operand1Exponent,operand2Exponent,resultExponent,
operand1NormalizedExponent,operand2NormalizedExponent;
rand bit [`w:0] intermediateExponent,intermediateNormalizedExponent,
roundExponent,roundNormalizedExponent;
rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
rand bit [`p-1:0] operand1Significand,operand2Significand,resultSignificand,
intermediateSignificand,intermediateNormalizedSignificand,
```

```
roundSignificand,roundNormalizedSignificand;
rand bit [`p-2:0] intermediateGuardSticky,intermediateNormalizedGuardSticky;
rand bit carry,roundCarry,roundValue;
rand bit intermediateGuard,intermediateRound,intermediateSticky;
rand bit intermediateNormalizedGuard,intermediateNormalizedRound,
intermediateNormalizedSticky;
rand int intermediateShiftValue,intermediateNormalizedShiftValue;
rand bit inexactFlag,underflowFlag;
const bit invalidFlag = 1'b0,overflowFlag = 1'b0;
const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
isOperand1Inf = 1'b0,isOperand2Inf = 1'b0;
const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b0,
isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;;
constraint binary_encoding_decoding {
{operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
{operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
{resultSign,resultExponent,resultMantissa} == result;
}
constraint is_operands_infinity_NaN_zero_subnormal {
if (isOperand1NaN)
((operand1Exponent == '1) && (operand1Mantissa != '0));
else if (isOperand1Inf)
((operand1Exponent == '1) && (operand1Mantissa == '0));
else
operand1Exponent != '1 && operand1Exponent != '0;
if (isOperand2NaN)
((operand2Exponent == '1) && (operand2Mantissa != '0));
else if (isOperand2Inf)
((operand2Exponent == '1) && (operand2Mantissa == '0));
else
operand2Exponent == '0;
}
// will vary in subnormal
constraint significand_mantissa  {
({1'b1,operand1Mantissa} == operand1Significand) &&
 (operand1NormalizedExponent == operand1Exponent) ;
({1'b0,operand2Mantissa} == operand2Significand) &&
(operand2NormalizedExponent == operand2Exponent) ;
}
function [2*`p-1:0] partial_product_multiplication(input [`p-1:0] op1,op2);
bit [2*`p-1:0] normal[0:`p-1], shifted[0:`p];
shifted[0] = 0;
for (int i = 0; i <= `p-1; i++)
begin
if (op2[i])
normal[i] = {`p'b0,op1};
else
normal[i] = '0;
shifted[i+1] = (normal[i] << i) + shifted[i];
end
```

```
return shifted[`p];
endfunction
constraint multiplication {
{carry,intermediateSignificand,intermediateGuardSticky} ==
partial_product_multiplication(operand1Significand,operand2Significand);
intermediateGuard == intermediateGuardSticky[`p-2];
intermediateRound == intermediateGuardSticky[`p-3];
intermediateSticky == (|intermediateGuardSticky[`p-4:0]);
intermediateSign == (operand1Sign ^ operand2Sign);
intermediateExponent == (operand1Exponent + operand2Exponent - `bias + 1);
}
function int leading_zero_calculation (input [0:`p+2] functionSignificand);
   for (int i = 0; i <= `p+2;i++)
     if (functionSignificand[i] == 1'b1) return i;
   return `p+3;
endfunction
constraint carry_correction {
intermediateSign == intermediateNormalizedSign;
intermediateShiftValue == leading_zero_calculation(intermediateSignificand);
if (intermediateShiftValue >= intermediateExponent)
 intermediateNormalizedShiftValue == intermediateExponent;
else intermediateNormalizedShiftValue == intermediateShiftValue;
(intermediateNormalizedExponent ==
(intermediateExponent - intermediateNormalizedShiftValue)) &&
({intermediateNormalizedSignificand,intermediateNormalizedGuardSticky} ==
 ({intermediateSignificand,intermediateGuardSticky} <<
 intermediateNormalizedShiftValue));
intermediateNormalizedGuard == intermediateNormalizedGuardSticky[`p-2];
intermediateNormalizedRound == intermediateNormalizedGuardSticky[`p-3];
intermediateNormalizedSticky == (|intermediateNormalizedGuardSticky[`p-4:0]);
}
constraint underflow_flag {
if (intermediateExponent > intermediateShiftValue) underflowFlag  == 1'b0;
else  underflowFlag == 1'b1;
}
constraint inexact_flag {
inexactFlag  ==  (intermediateNormalizedGuard  |  intermediateNormalizedRound  |
intermediateNormalizedSticky | underflowFlag);
}
constraint rounding {
(roundDirection == roundZero) -> (roundValue == 1'b0);
((underflowFlag == 1'b0) && (roundDirection == roundPositive)) ->
(roundValue  ==  (~intermediateNormalizedSign  &  (intermediateNormalizedGuard  |
intermediateNormalizedRound | intermediateNormalizedSticky)));
((underflowFlag == 1'b0) && (roundDirection == roundNegative)) ->
(roundValue  ==  (intermediateNormalizedSign  &  (intermediateNormalizedGuard  |
intermediateNormalizedRound | intermediateNormalizedSticky)));
((underflowFlag == 1'b0) && (roundDirection == roundTieEven)) ->
(roundValue == (intermediateNormalizedGuard & (intermediateNormalizedRound |
intermediateNormalizedSticky | intermediateNormalizedSignificand[0])));
```

```
((underflowFlag == 1'b1) && (roundDirection == roundPositive)) ->
(roundValue == (~intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedRound | intermediateNormalizedSticky |
intermediateNormalizedSignificand[0])));
((underflowFlag == 1'b1) && (roundDirection == roundNegative)) ->
(roundValue == (intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedRound | intermediateNormalizedSticky |
intermediateNormalizedSignificand[0])));
((underflowFlag == 1'b1) && (roundDirection == roundTieEven)) ->
(roundValue == (intermediateNormalizedSignificand[0] &
(intermediateNormalizedGuard | intermediateNormalizedRound |
intermediateNormalizedSticky | intermediateNormalizedSignificand[1])));
}
constraint addition_after_round {
roundSign == intermediateNormalizedSign;
if (underflowFlag)
{roundCarry,roundSignificand} == ({2'b0,intermediateNormalizedSignificand[`p-
1:1]} + roundValue) &&
roundExponent == '0;
else
{roundCarry,roundSignificand} == ({1'b0,intermediateNormalizedSignificand} +
roundValue) &&
(roundExponent == intermediateNormalizedExponent);
}
constraint normalization_after_rounding {
roundNormalizedSign == roundSign;
(roundCarry == 1'b1) -> (
(roundNormalizedExponent == roundExponent + 1) &&
(roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
(roundCarry == 1'b0) -> (
(roundNormalizedExponent == roundExponent) &&
(roundNormalizedSignificand == roundSignificand));
}
constraint result_calculation {
(resultSign == roundNormalizedSign);
(resultExponent == roundNormalizedExponent);
(resultSignificand == roundNormalizedSignificand);
(resultMantissa == roundNormalizedSignificand[`p-2:0]);
}
endclass
int i;
floating_point_numbers_variables fpv;
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
```

```
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);

$display("Flags : %b %b %b %b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
```

Appendix C: SV constraints for Division

```
`define N 1000  //Number of generated test vectors
`define k 32 //Change to 64 for double
`define p 24 //Change to 53 for double
`define w 8 //Change to 11 for double
`define emax 127 //Change to 1023 for double
`define bias 127 //Change to 1023 for double
module DUT_normal;
class floating_point_numbers_variables;
rand bit [`k-1:0] operand1,operand2,result;
typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
rand roundTypes roundDirection;
rand bit operand1Sign,operand2Sign,resultSign,
intermediateSign,intermediateNormalizedSign,
roundSign,roundNormalizedSign;
rand bit [`w-1:0] operand1Exponent,operand2Exponent,resultExponent,
operand1NormalizedExponent,operand2NormalizedExponent;
rand bit [`w:0] intermediateExponent,intermediateNormalizedExponent,
roundExponent,roundNormalizedExponent;
rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
rand bit [`p-1:0] operand1Significand,operand2Significand,resultSignificand,
intermediateSignificand,intermediateNormalizedSignificand,
roundSignificand,roundNormalizedSignificand;
rand bit [`p-2:0] intermediateGuardSticky;
rand bit roundCarry,roundValue;
rand bit intermediateGuard,intermediateSticky;
rand bit intermediateNormalizedGuard,intermediateNormalizedSticky;
rand bit inexactFlag,overflowFlag,underflowFlag;
const bit invalidFlag = 1'b0;
const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
isOperand1Inf = 1'b0,isOperand2Inf = 1'b0;
const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b0,
isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;;
rand int intermediateNormalizedShiftValue;
constraint binary_encoding_decoding {
{operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
{operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
{resultSign,resultExponent,resultMantissa} == result;
}
constraint is_operands_infinity_NaN_zero_subnormal {
if (isOperand1NaN)
((operand1Exponent == '1) && (operand1Mantissa != '0));
else if (isOperand1Inf)
((operand1Exponent == '1) && (operand1Mantissa == '0));
else
operand1Exponent != '1 && operand1Exponent != '0;
if (isOperand2NaN)
```

```
((operand2Exponent == '1) && (operand2Mantissa != '0));
else if (isOperand2Inf)
((operand2Exponent == '1) && (operand2Mantissa == '0));
else
operand2Exponent != '1 && operand2Exponent != '0;
}
constraint significand_mantissa  {
({1'b1,operand1Mantissa} == operand1Significand) &&
(operand1NormalizedExponent == operand1Exponent) ;
({1'b1,operand2Mantissa} == operand2Significand) &&
(operand2NormalizedExponent == operand2Exponent) ;
}
function [`p-1:-`m] iterative_div(input [`p-1:0] divident,divisor);
bit [2*`p-1:-`m] r[-`p+1:`p], d[-`p+1:`p];
bit [`p-1:-`m] q[-`p+1:`p];
r[-`p+1][2*`p-1:`p] = divident;
q[-`p+1] = '0;
d[-`p+1][2*`p-1:`p] = divisor;
for (int i = -`p+2; i <= `p; i++)
begin
if (r[i-1] >= d[i-1])
begin
r[i] = r[i-1] - d[i-1];
q[i] = {q[i-1][`p-2:-`m],1'b1};
end
else
begin
r[i] = r[i-1];
q[i] = {q[i-1][`p-2:-`m],1'b0};
end
d[i] = {1'b0,d[i-1][2*`p-1:-`m+1]};
end
return q[`p];
endfunction
constraint division {
{intermediateSignificand,intermediateGuardSticky} ==
iterative_div(operand1Significand,operand2Significand);
intermediateSign == (operand1Sign ^ operand2Sign);
intermediateExponent == (operand1Exponent - operand2Exponent + `bias);
intermediateGuard ==  intermediateGuardSticky[`p-2];
intermediateSticky == |intermediateGuardSticky[`p-3:0];
}
constraint underflow_flag {
if (intermediateExponent > 0 && intermediateSignificand[`p-1] == 1'b1)
 underflowFlag  == 1'b0;
else if (intermediateExponent > 1 && intermediateSignificand[`p-1] == 1'b0)
underflowFlag  == 1'b0;
else  underflowFlag == 1'b1;
}
constraint normalization {
```

```
intermediateSign == intermediateNormalizedSign;
if (intermediateSignificand[`p-1] == 1'b1 && underflowFlag == 1'b0)
(intermediateNormalizedExponent == (intermediateExponent)) &&
(intermediateNormalizedGuard == intermediateGuard) &&
(intermediateNormalizedSticky == intermediateSticky) &&
(intermediateNormalizedSignificand == intermediateSignificand);
else if (intermediateSignificand[`p-1] == 1'b0 && underflowFlag == 1'b1)
(intermediateNormalizedExponent == (intermediateExponent - 1)) &&
(intermediateNormalizedGuard == intermediateGuard) &&
(intermediateNormalizedSticky == intermediateSticky) &&
(intermediateNormalizedSignificand == intermediateSignificand);
else
(intermediateNormalizedExponent == intermediateExponent - 1) &&
(intermediateNormalizedGuard == intermediateGuardSticky[`p-3]) &&
(intermediateNormalizedSticky == |intermediateGuardSticky[`p-4:0]) &&
(intermediateNormalizedSignificand ==
{intermediateSignificand[`p-2:0],intermediateGuard});

}
constraint overflow_flag {
if (intermediateNormalizedExponent < (`bias+`bias+1)) overflowFlag == 1'b0;
else overflowFlag == 1'b1;
}
constraint inexact_flag {
inexactFlag == (intermediateNormalizedGuard | intermediateNormalizedSticky |
overflowFlag | underflowFlag);
}
constraint rounding {
(roundDirection == roundZero) -> (roundValue == 1'b0);
(roundDirection == roundPositive) ->
(roundValue == (~intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedSticky)));
(roundDirection == roundNegative) ->
 (roundValue == (intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedSticky)));
(roundDirection == roundTieEven) ->
 (roundValue == (intermediateNormalizedGuard & (intermediateNormalizedSticky |
intermediateNormalizedSignificand[0])));
}
constraint addition_after_round {
{roundCarry,roundSignificand} == ({1'b0,intermediateNormalizedSignificand} +
roundValue);
roundSign == intermediateNormalizedSign;
roundExponent == intermediateNormalizedExponent;
}
constraint normalization_after_rounding {
roundNormalizedSign == roundSign;
(roundCarry == 1'b1) -> (
(roundNormalizedExponent == roundExponent + 1) &&
(roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
```

```
(roundCarry == 1'b0) -> (
(roundNormalizedExponent == roundExponent) &&
(roundNormalizedSignificand == roundSignificand));
}
constraint result_calculation {
if (overflowFlag && (roundDirection == roundTieEven))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0);
else if (overflowFlag && ((roundDirection == roundZero) || ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b1)) || ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b0))))
(resultSign == roundNormalizedSign) &&
(resultExponent == `bias+`bias) &&
(resultSignificand == '1) &&
(resultMantissa == '1);
else if (overflowFlag && ((roundDirection == roundTieEven) || ((roundDirection == roundPositive) && (roundNormalizedSign == 1'b0)) || ((roundDirection == roundNegative) && (roundNormalizedSign == 1'b1))))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0);
else
(resultSign == roundNormalizedSign) &&
(resultExponent == roundNormalizedExponent[`w-1:0]) &&
(resultSignificand == roundNormalizedSignificand) &&
(resultMantissa == roundNormalizedSignificand[`p-2:0]);
}
endclass
int i;
floating_point_numbers_variables fpv;
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);
$display("Flags : %b %b %b %b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
module DUT_subnormal;
class floating_point_numbers_variables;
rand bit [`k-1:0] operand1,operand2,result;
```

```
typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
rand roundTypes roundDirection;
rand bit operand1Sign,operand2Sign,resultSign,
intermediateSign,intermediateNormalizedSign,roundSign,roundNormalizedSign;
rand bit [`w-1:0] operand1Exponent,operand2Exponent,resultExponent,
operand2NormalizedExponent;
rand bit [`w:0] intermediateExponent,intermediateNormalizedExponent,
roundExponent,roundNormalizedExponent;
rand bit [`p-2:0] operand1Mantissa,operand2Mantissa,resultMantissa;
rand bit [`p-1:0]
operand1Significand,operand2Significand,operand2NormalizedSignificand,
resultSignificand,intermediateSignificand,intermediateNormalizedSignificand,round
Significand,roundNormalizedSignificand;
rand bit [`p-2:0] intermediateGuardSticky;
rand bit roundCarry,roundValue;
rand bit intermediateGuard,intermediateSticky;
rand bit intermediateNormalizedGuard,intermediateNormalizedSticky;
rand bit inexactFlag,overflowFlag,underflowFlag;
const bit invalidFlag = 1'b0;
const bit isOperand1NaN = 1'b0,isOperand2NaN = 1'b0,
isOperand1Inf = 1'b0,isOperand2Inf = 1'b0;
const bit isOperand1Subnormal = 1'b0,isOperand2Subnormal = 1'b1,
isResultSubnormal = 1'b0,isOperand1Zero = 1'b0,isOperand2Zero = 1'b0;;
rand int exponent_correction;
constraint binary_encoding_decoding {
{operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
{operand2Sign,operand2Exponent,operand2Mantissa} == operand2;
{resultSign,resultExponent,resultMantissa} == result;
}
constraint is_operands_infinity_NaN_zero_subnormal {
if (isOperand1NaN)
((operand1Exponent == '1) && (operand1Mantissa != '0));
else if (isOperand1Inf)
((operand1Exponent == '1) && (operand1Mantissa == '0));
else
operand1Exponent != '1 && operand1Exponent != '0;
if (isOperand2NaN)
((operand2Exponent == '1) && (operand2Mantissa != '0));
else if (isOperand2Inf)
((operand2Exponent == '1) && (operand2Mantissa == '0));
else
operand2Exponent == '0;
}
// will vary in subnormal
constraint significand_mantissa  {
({1'b1,operand1Mantissa} == operand1Significand) ;
({1'b0,operand2Mantissa} == operand2Significand) ;
}
function int leading_zero_calculation (input [0:`p-1] functionSignificand);
for (int i = 0;i < `p;i++) begin
```

```
if (functionSignificand[i] == 1'b1) return i;
end
endfunction
constraint divisor_normalized {
exponent_correction == leading_zero_calculation(operand2Significand);
operand2NormalizedSignificand == operand2Significand << exponent_correction;
}
function [`p-1:-`m] iterative_div(input [`p-1:0] divident,divisor);
bit [2*`p-1:-`m] r[-`p+1:`p], d[-`p+1:`p];
bit [`p-1:-`m] q[-`p+1:`p];
r[-`p+1][2*`p-1:`p] = divident;
q[-`p+1] = '0;
d[-`p+1][2*`p-1:`p] = divisor;
for (int i = -`p+2; i <= `p; i++)
begin
if (r[i-1] >= d[i-1])
begin
r[i] = r[i-1] - d[i-1];
q[i] = {q[i-1][`p-2:-`m],1'b1};
end
else
begin
r[i] = r[i-1];
q[i] = {q[i-1][`p-2:-`m],1'b0};
end
d[i] = {1'b0,d[i-1][2*`p-1:-`m+1]};
end
return q[`p];
endfunction

constraint division {
{intermediateSignificand,intermediateGuardSticky} ==
iterative_div(operand1Significand,operand2NormalizedSignificand);
intermediateSign == (operand1Sign ^ operand2Sign);
intermediateExponent == (operand1Exponent + exponent_correction + `bias-1);
intermediateGuard ==  intermediateGuardSticky[`p-2];
intermediateSticky == |intermediateGuardSticky[`p-3:0];
}
constraint underflow_flag {
if (intermediateExponent > 0 && intermediateSignificand[`p-1] == 1'b1)
 underflowFlag  == 1'b0;
else if (intermediateExponent > 1 && intermediateSignificand[`p-1] == 1'b0)
 underflowFlag  == 1'b0;
else  underflowFlag == 1'b1;
}
constraint normalization {
intermediateSign == intermediateNormalizedSign;
if (intermediateSignificand[`p-1] == 1'b1 && underflowFlag == 1'b0)
(intermediateNormalizedExponent == (intermediateExponent)) &&
(intermediateNormalizedGuard == intermediateGuard) &&
```

```
(intermediateNormalizedSticky == intermediateSticky) &&
(intermediateNormalizedSignificand == intermediateSignificand);
else if (intermediateSignificand[`p-1] == 1'b0 && underflowFlag == 1'b1)
(intermediateNormalizedExponent == (intermediateExponent - 1)) &&
(intermediateNormalizedGuard == intermediateGuard) &&
(intermediateNormalizedSticky == intermediateSticky) &&
(intermediateNormalizedSignificand == intermediateSignificand);
else
(intermediateNormalizedExponent == intermediateExponent - 1) &&
(intermediateNormalizedGuard == intermediateGuardSticky[`p-3]) &&
(intermediateNormalizedSticky == |intermediateGuardSticky[`p-4:0]) &&
(intermediateNormalizedSignificand ==
{intermediateSignificand[`p-2:0],intermediateGuard});
}
constraint overflow_flag {
if (intermediateNormalizedExponent < (`bias+`bias+1)) overflowFlag == 1'b0;
else overflowFlag == 1'b1;
}
constraint inexact_flag {
inexactFlag == (intermediateNormalizedGuard | intermediateNormalizedSticky |
overflowFlag | underflowFlag);
}
constraint rounding {
(roundDirection == roundZero) -> (roundValue == 1'b0);
(roundDirection == roundPositive) ->
 (roundValue == (~intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedSticky)));
(roundDirection == roundNegative) ->
(roundValue == (intermediateNormalizedSign & (intermediateNormalizedGuard |
intermediateNormalizedSticky)));
(roundDirection == roundTieEven) ->
(roundValue == (intermediateNormalizedGuard & (intermediateNormalizedSticky |
intermediateNormalizedSignificand[0])));
}
constraint addition_after_round {
{roundCarry,roundSignificand} ==
({1'b0,intermediateNormalizedSignificand} + roundValue);
roundSign == intermediateNormalizedSign;
roundExponent == intermediateNormalizedExponent;
}
constraint normalization_after_rounding {
roundNormalizedSign == roundSign;
(roundCarry == 1'b1) -> (
(roundNormalizedExponent == roundExponent + 1) &&
(roundNormalizedSignificand == {1'b1,roundSignificand[`p-1:1]}));
(roundCarry == 1'b0) -> (
(roundNormalizedExponent == roundExponent) &&
(roundNormalizedSignificand == roundSignificand));
}
constraint result_calculation {
```

```
if (overflowFlag && (roundDirection == roundTieEven))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0);
else if (overflowFlag && ((roundDirection == roundZero) || ((roundDirection ==
roundPositive) && (roundNormalizedSign == 1'b1)) || ((roundDirection ==
roundNegative) && (roundNormalizedSign == 1'b0))))
(resultSign == roundNormalizedSign) &&
(resultExponent == `bias+`bias) &&
(resultSignificand == '1) &&
(resultMantissa == '1);
else if (overflowFlag && ((roundDirection == roundTieEven) || ((roundDirection ==
roundPositive) && (roundNormalizedSign == 1'b0)) || ((roundDirection ==
roundNegative) && (roundNormalizedSign == 1'b1))))
(resultSign == roundNormalizedSign) &&
(resultExponent == '1) &&
(resultSignificand == {1'b1,'0}) &&
(resultMantissa == '0);
else
(resultSign == roundNormalizedSign) &&
(resultExponent == roundNormalizedExponent[`w-1:0]) &&
(resultSignificand == roundNormalizedSignificand) &&
(resultMantissa == roundNormalizedSignificand[`p-2:0]);
}
endclass
int i;
floating_point_numbers_variables fpv;
initial
begin
i = 0;
fpv = new();
repeat (`N) begin
assert(fpv.randomize());
i++;
$display("Test ID: %d ",i);
$display("Test vector: %b %b %b",fpv.operand1,fpv.operand2,fpv.result);
$display("Flags : %b %b %b
%b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
end
end
endmodule
```

Appendix D: SV constraints for Square Root

```
`define N 1000  //Number of generated test vectors
`define k 32 //Change to 64 for double
`define p 24 //Change to 53 for double
`define w 8 //Change to 11 for double
`define emax 127 //Change to 1023 for double
`define bias 127 //Change to 1023 for double
module DUT;
 class floating_point_numbers_variables;
 rand bit [`k-1:0] operand1,result;
 typedef enum {roundTieEven,roundPositive,roundNegative,roundZero} roundTypes;
 rand roundTypes roundDirection;
 const bit operand1Sign = 1'b0,
 resultSign = 1'b0,intermediateSign = 1'b0,
 intermediateNormalizedSign = 1'b0,roundSign = 1'b0;
 rand bit [`w-1:0] operand1Exponent,resultExponent;
 rand bit [`w:0] intermediateExponent, intermediateNormalizedExponent,
 roundExponent;
 rand bit [`p-2:0] operand1Mantissa,resultMantissa;
 rand bit [`p-1:0] operand1Significand,resultSignificand,roundSignificand;
 rand bit [`p:0] intermediateSignificand;
 rand bit [`p-1:0] intermediateNormalizedSignificand;
 rand bit [`p+1:0] intermediateSignificandRoundSticky;
 rand bit roundValue;
 rand bit intermediateNormalizedRound,intermediateNormalizedSticky;
 rand bit inexactFlag;
 const bit overflowFlag = 1'b0, underflowFlag = 1'b0, invalidFlag = 1'b0;
 const bit isOperand1NaN = 1'b0,isOperand1Inf = 1'b0;
 const bit isOperand1Subnormal = 1'b0,isResultSubnormal = 1'b0,
 isOperand1Zero = 1'b0;
 const int half_bias = (`bias-1)/2;
 constraint binary_encoding_decoding {
  {operand1Sign,operand1Exponent,operand1Mantissa} == operand1;
  {resultSign,resultExponent,resultMantissa} == result;
 }
 constraint is_operands_infinity_NaN_zero_subnormal {
  if (isOperand1NaN)
   ((operand1Exponent == '1) && (operand1Mantissa != '0));
  else if (isOperand1Inf)
   ((operand1Exponent == '1) && (operand1Mantissa == '0));
  else
   operand1Exponent != '1 && operand1Exponent != '0;
 }
 constraint significand_mantissa  {
  ({1'b1,operand1Mantissa} == operand1Significand);
 }
 function [`p+1:0] sqrt(input [`p+1:0] i1);
```

```
 logic [`p+1:0] R,Q;
 logic R_;
 Q = {i1,`p'b0,2'b0} ** 0.5;
 R  = i1 - (Q ** 2);
 if (R == i1)
  R_ = 0;
 else
  R_ = 1;
 return {Q[`p+1:1],R_};
endfunction
function [`p+1:0] sqrt_iterative(input [`p+1:0] i1);
 logic [`p+2:0] F,F_t_1;
 logic [`p+1:0] R,Q;
 logic [`p+1:0] R_F;
 logic [1:0] temp;
 int i;
 i = 2*`p+2;
 F = 0; R = {i1,`p'b0,2'b0} >> i; Q = 0;
 for (int t = 1; t <= `p+1; t++) begin
  F_t_1 = F;
  i = i -2;
  temp = {i1,`p'b0,2'b0} >> i;
  if (R >= {F_t_1[`p:0],1'b1}) begin
   Q = {Q[`p:0],1'b1};
   F = ((F_t_1+F_t_1[0]) << 1) +1 ;
   R_F = R- F;
   R = (R_F <<2) +temp;
  end
  else begin
   Q = {Q[`p:0],1'b0};
   F = (F_t_1+F_t_1[0]) << 1 ;
   R = (R << 2) + temp;
  end
 end
 return Q;
endfunction
constraint sqaure_root {
 intermediateExponent == operand1Exponent[`w-1:1] + half_bias +
 operand1Exponent[0];
 if (operand1Exponent[0])
  intermediateSignificand == {1'b0,operand1Significand};
 else
  intermediateSignificand == {1'b0,operand1Significand} << 1;
  intermediateSignificandRoundSticky == sqrt1({intermediateSignificand,1'b0});
  intermediateNormalizedExponent == intermediateExponent;
  intermediateNormalizedSignificand ==
 {intermediateSignificandRoundSticky[`p+1:2]};
  intermediateNormalizedRound == intermediateSignificandRoundSticky[1];
  intermediateNormalizedSticky == intermediateSignificandRoundSticky[0];
}
```

```
 constraint inexact_flag {
  inexactFlag == (intermediateNormalizedRound || intermediateNormalizedSticky);
 }
 constraint rounding {
  (roundDirection == roundZero) -> (roundValue == 1'b0);
  (roundDirection == roundPositive) ->
  (roundValue == (~intermediateNormalizedSign &
  (intermediateNormalizedRound | intermediateNormalizedSticky)));
  (roundDirection == roundNegative) ->
  (roundValue == (intermediateNormalizedSign &
  (intermediateNormalizedRound  | intermediateNormalizedSticky)));
  (roundDirection == roundTieEven) ->
  (roundValue == (intermediateNormalizedRound &
  (intermediateNormalizedSticky | intermediateNormalizedSignificand[0])));
 }
 constraint addition_after_round {
  roundSignificand == intermediateNormalizedSignificand + roundValue;
  roundExponent == intermediateNormalizedExponent;
 }
 constraint result_calculation {
  (resultExponent == roundExponent[`w-1:0]) &&
  (resultSignificand == roundSignificand) &&
  (resultMantissa == roundSignificand[`p-2:0]);
 }
 endclass
 int i;
 floating_point_numbers_variables fpv;
 initial
  begin
   i = 0;
   fpv = new();
   repeat (`N) begin
   assert(fpv.randomize());
   i++;
   $display("Test ID: %d ",i);
   $display("Test vector: %b %b",fpv.operand1,fpv.result);
   $display("Flags                                    :          %b          %b          %b
%b",fpv.inexactFlag,fpv.overflowFlag,fpv.underflowFlag,fpv.invalidFlag);
  end
 end
endmodule
```

الملخص

التحقق من عمليات النقطة العائمة مهمة صعبة المنال، وتكلفة علاج الخطأ في مرحلة ما بعد الإنتاج قاسية. هذا بسبب التعامل مع مدخلات ثنائية كبيرة ومن ثم فشل التحقق المبني على المحاكاة لتغطية كافة المدخلات الممكنة، وبالتالي لا ضمان لعدم وجود خلل في التصميم. من ناحية أخرى، اثبتت الطرق المبنية على اساس رياضي الفاعلية في التحقق من هذا المجال، إلا أنها تحتاج إلى خلق نموذج رياضي للتصميم، ايضا لا يمكن أن تعمل على نسخة معدلة من تصميم وربما تفشل مع تصاميم معقدة بسبب كبر حجم النموزج.

لدينا مقترح جديدا للتحقق من صحة العمليات الثنائية للنقطة العائمة بإستخدام تقنية توليد اختبارات عشوائية مقيدة. يتم كتابة القيود المستخدمة في التحقق لدينا بإستخدام لغة معيارية(System Verilog) ويمكن حلها مع أي أداة للمحاكاه تدعم هذه اللغة المعيارية. لكل عملية حسابية، يتم كتابة قيود تربط بين المدخلات، النتائج المتوسطة، التقريب، و النتيجة النهائية ليتوافق مع المواصفة المعيارية IEEE-754.

الاقتراح الجديد هو عام، ويمكن استخدامه للتحقق من أي برنامج أو جهاز يقوم بحساب العمليات الثنائية للنقطة العائمة. أيضا، فإنه يثبت جدوى وفائدة المقترح في العثور على أخطاء في مختلف العمليات الحسابية الثنائية للنقطة العائمة.

مهندس: خالد محمد عبد المقصود نوح

تاريخ الميلاد: 1987\07\20

الجنسية: مصري

تاريخ التسجيل: 2010\10\1

تاريخ المنح: 2016

القسم: هندسة الإلكترونيات والاتصالات الكهربية

الدرجة: ماجستير العلوم

المشرفون:

أ.د. حسام علي حسن فهمي

الممتحنون:

أ.د. حسام علي حسن فهمي

أ.د. إبراهيم محمد قمر

أ.د. أشرف محمد سالم، كلية الهندسة، جامعة عين شمس

عنوان الرسالة:

التحقق من الحسابات الثنائية ذات النقطة العائمة بإستخدام لغة معيارية لحل القيود

الكلمات الدالة:

عمليات النقطة العائمة ، المحاكاة المقيدة

ملخص الرسالة:

التحقق من عمليات النقطة العائمة مهمة صعبة المنال، وتكلفة علاج الخطأ في مرحلة ما بعد الإنتاج قاسية. هذا بسبب التعامل مع مدخلات ثنائية كبيرة ومن ثم فشل التحقق المبني على المحاكاة لتغطية كافة المدخلات الممكنة، وبالتالي لا ضمان لعدم وجود خلل في التصميم. من ناحية أخرى، اثبتت الطرق المبنية على اساس رياضي الفاعلية في التحقق من هذا المجال، إلا أنها تحتاج إلى خلق نموذج رياضي للتصميم، ايضا لا يمكن أن تعمل على نسخة معدلة من تصميم وربما تفشل مع تصاميم معقدة بسبب كبر حجم النموزج. لدينا مقترح جديدا للتحقق من صحة العمليات الثنائية للنقطة العائمة بإستخدام تقنية توليد اختبارات عشوائية مقيدة. يتم كتابة القيود المستخدمة في التحقق لدينا بإستخدام لغة معيارية(System Verilog) ويمكن حلها مع أي أداة للمحاكاه تدعم هذه اللغة المعيارية. لكل عملية حسابية، يتم كتابة قيود تربط بين المدخلات، النتائج المتوسطة، التقريب، و النتيجة النهائية ليتوافق مع المواصفة المعيارية IEEE-754.

التحقق من الحسابات الثنائية ذات النقطة العائمة بإستخدام لغة معيارية لحل القيود

اعداد
خالد محمد عبد المقصود نوح

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الإلكترونيات والاتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور: حسام علي حسن فهمي المشرف الرئيسى

الاستاذ الدكتور: إبراهيم محمد قمر الممتحن الداخلي

الاستاذ الدكتور: أشرف محمد سالم الممتحن الخارجي، كلية الهندسة، جامعة عين شمس

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2016

التحقق من الحسابات الثنائية ذات النقطة العائمة بإستخدام لغة معيارية لحل القيود

اعداد
خالد محمد عبد المقصود نوح

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الإلكترونيات والاتصالات الكهربية

تحت اشراف

ا. د. حسام علي حسن فهمى
قسم االلكترونيات و الاتصالات
كلية الهندسة جامعة القاهرة

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2016

التحقق من الحسابات الثنائية ذات النقطة العائمة بإستخدام لغة معيارية لحل القيود

اعداد

خالد محمد عبد المقصود نوح

رسالة مقدمة إلى كلية الهندسة - جامعة القاهرة
كجزء من متطلبات الحصول على درجة ماجستيرالعلوم
في
هندسة الإلكترونيات والاتصالات الكهربية

كليــة الهندســة - جامعــة القاهــرة
الجيـزة - جمهوريـة مصـر العربيــة

2016