Design and Verification of a multi-mode floating point conversion IP using SystemVerilog

A Thesis
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY

Utkarsh Gupta

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Prof. Gerald Sobelman

September 2015

Acknowledgements

I would like to express my sincere gratitude to my advisor, Professor Gerald Sobelman, for the continuous support and guidance during my research work. This would have not been possible without his patience, motivation and knowledge. His direction helped me during my research and also during writing of this thesis and completing my Master's thesis efficiently.

I would also like to thank my committee members, Prof. Kiarash Bazargan and Prof. Antonia Zhai for willing to be a part of my thesis committee and review panel and providing insightful comments on my thesis. Lastly, I would like to thank my family and friends for their constant support and inspiration.

Dedication

This thesis is dedicated to my family and friends.

Abstract

This research work focuses on a hardware level implementation and verification of a multi-mode floating point conversion IP (Intellectual Property) capable of converting from the various floating point formats to the fixed-point format and vice-versa. This conversion can help improve the performance of a complex system with respect to speed, power consumption and cost and would also help in the design of an embedded system, where mostly fixed-point formats are used during the design, as it allows drastic savings in all traditional cost metrics. The proposed IP can work alongside a main processor or master which would be the main processing unit and would accept the input data and convert it to the required format and send it to the main processor. The results from the main processor can be fed again to the IP to convert it to the desired floating or fixed point format. The suggested design also includes the recently introduced half-precision floating point format (16-bit), which because of its advantages over the traditional single and double precision formats is finding use in many applications. All the conversion modules are compliant with the IEEE 754-2008 standard and also include the rounding modes and exception signals. The design is then verified using the concepts of Verification Methodology Manual (VMM) standard. The design and verification is performed using the SystemVerilog language.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Overview

Majority of the real world applications mainly use two number formats to store and manipulate the numeric representation of data namely, fixed point and floating point. Various types of architectures have been developed compliant with both the above formats, each type having its benefits and drawbacks. Many applications like weather forecasting, image processing, linear algebraic routines, robotics and digital signal processing also involve conversions from one format to the other to improve performance and efficiency [2]. Typically, these conversions are performed as instructions in microprocessors, which complicate the architecture of the internal processor core [2]. This research work focuses on a hardware design and verification of a conversion IP between the various number formats. This IP could be used alongside a host processor which would focus on the computational aspect, whereas the IP would enable the desired conversions, improving the system performance. The IP is designed to convert between the various floating point formats (single precision, double precision and half precision) and also between floating point and fixed point formats. The floating point units designed are compliant with the IEEE 754-2008 standard [1]. The full design of the conversion IP is specified using SystemVerilog and is verified using the concepts of Verification Methodology Manual (VMM).

1.2 Floating point vs. Fixed point

Floating and fixed point represent the two ways the numeric data can be represented and manipulated in digital signal processing. Each technique has its advantages and disadvantages. The term floating and fixed refers to the position of the decimal point while representing a real number. In case of a fixed point representation, the number of bits after the decimal point is fixed, whereas a floating point representation resembles the scientific notation and the number is specified in terms of the exponent and significand. For example, considering the decimal (base 10) representation, a fixed point number may be specified as 567.89, 56.78, 5.67, etc. whereas in case of a floating point number we may have 5.6789, 5678.9, 0.0056789, 56789000 etc. Thus, the range of numbers represented in floating point is much greater as compared to fixed point representation with equal number of bits. Also, the precision for fixed point is definite between two adjacent numbers, but in case of floating point the precision is dynamic, i.e. for small numbers the gap is small and for larger numbers the gap is large.

Considering speed of execution, fixed point is much faster as compared to floating point, in case of general processors, however in case of digital signal processors the difference in speed is less owing to the highly optimized hardware for floating point operations. The architecture needed to support floating point formats is highly complicated as the adder and multiplier unit need to be specially optimized. The instruction set also is larger as it needs to support fixed point operations as well. In case of fixed point, the architectures are simple resulting in higher efficiency in terms of power and performance. This also results in fixed point processors being cheaper as compared to their floating point counterparts.

On the other hand, the product development cycle is shorter for floating point as compared to fixed point as most of the algorithms are developed considering floating point representations. This is because the programmer doesn't have to worry about issues such as overflow, underflow, rounding error, etc. which would arise in case of fixed point representations. The floating point processors also offer high computational power with greater accuracy and precision. Thus there is always a trade-off between the choice of using a fixed point or floating point processor depending on the application and data set. The proposed IP would help in applications where conversions are needed between fixed and floating number representations, in large complex systems or algorithmic codes. These conversions are mainly available in software as algorithms or instructions; however a hardware unit would help improve the speed remarkably and also help simplify the internal core architecture which would further help in increasing the system performance and efficiency. For this research, the major focus has been on the SystemVerilog design of the IP which is implemented as a pipelined architecture and its verification using the concepts of VMM.

1.3 The proposed IP

The IP is designed in a modular way, where each module is responsible for a particular conversion. Each module is pipelined to improve throughput. The processor can accept large amount of input data and depending on the op-code or desired operation, the input data can be converted to the required number format. The supported operations include conversions among the various floating point formats specified in the IEEE 754-2008 standard. These include the single precision, double precision and the half precision formats. It is also capable of converting each of the floating point formats to the fixed point format. The fixed point format is represented as 32 bits in the Q17.15 format i.e. 17 bits before and 15 bits after the decimal point. The signed numbers are represented in the two's complement representation. The design also supports four rounding modes i.e. round ties to even, round towards positive infinity, round towards negative infinity and round towards zero. It also supports the overflow, underflow, inexact and invalid exceptions and they are handled according to the IEEE standard.

1.4 Contribution of the thesis

In [2] a similar type of IP was suggested which would convert between floating and fixed point formats. This thesis, has contributed to that work by adding a new floating point format which has been recently added to the IEEE 754-2008 standard, namely half precision. This new format is represented by 16 bits, i.e. half of the conventional single precision format (32 bits). It would be useful in many application areas, where the large range of single or double precision is not desired. It would require half the storage space and half the memory bandwidth compared to the single precision representation. Even though it would have a smaller range, it would be ideal for storing floating point values in many situations where precision isn't critical. It was recently proposed in [13] that many applications which use floating point representation like speech recognition, computational fluid dynamics and shock hydrodynamics use larger number of bits as required for the desired accuracy and precision. Using larger number of bits causes loss in performance as it consumes more storage space and consumes more power. Replacing the traditional floating point format with half-precision would help improve performance in many ways. The proposed IP includes conversion from single and double precision floating point representations to half precision floating point representation and vice-versa. It also includes conversions between half precision and fixed point formats. This acts as a major addition to the previous work done in the area of floating and fixed point conversions. The advantages of the half-precision floating point format can be summarized as follows,

1. It can act as a useful format for storing floating point numbers because it requires half the storage space as compared to the traditional single precision floating point format which occupies 32-bits.

2. It would also require half the memory bandwidth as compared to the single precision floating point format, hence reduce the memory bus traffic, which causes a serious bottleneck in many applications.

3. Better performance in applications that load and store floating point values in certain scenarios where 16-bit precision is sufficient.

4. Unlike the 8 or 16-bit integer formats, half floats still possess the benefit of having a dynamic range and precision, meaning they have relatively high precision for floating point values near zero, but have low precision for integers far from zero.

5. Since they are half the size, they fit into the lower level cache with lower latency and take up half the cache space, which frees up the cache space for other data in your program, thereby improving performance.

1.5 Motivation

Many applications in fields like weather forecasting, robotics, digital signal processing and image processing use the floating point formats to represent and manipulate data [2]. These applications deal with floating numbers and hence have complicated architectures and also consume a lot of power. These issues can be solved by using the IP and using fixed-point arithmetic, where the range of the data set is small and high precision is not required.

1. In image processing, applications like feature extraction and segmentation requires large floating point matrix multiplication. These multiplications are time consuming and also require a lot of power [2]. Hence, the IP can be used to convert the operands to fixed-point and then perform the arithmetic operation. The arithmetic operations are much more complicated in the case of floating point as the exponent and significand have to be dealt separately, unlike the fixed-point arithmetic which is simple. The result can be converted back into the floating point format, using the IP.

2. Operations like FFT and DFT in digital signal processing usually deal with fixed point numbers. If the coefficients or the input to these blocks are in floating point format and do not require the large range and precision of floating point numbers, the IP can convert them into fixed point which would make the computation simpler and faster.

3. In case of embedded systems, there is an increased pressure in terms of rapidly increasing performance and reducing power consumption. During the end stage of the design, both the software and the hardware use only fixed-point numeric formats, because this allows tremendous savings in all cost metrics like required silicon area, power consumption and performance of the final implementation [5]. Thus for any data set which include floating point formats, they need to be converted to fixed-point. This conversion can be performed by the proposed conversion IP.

4. In video applications, the sampling rate is high leading to tens or hundreds of megabits per second in pixel data. Pixel data is usually represented in three words, one for each of the red, green and blue (RGB) planes of the image. In most systems each color requires 8 to 12 bits. Mathematical operations use the MPEG video compression algorithms including DCTs and quantization and there is limited filtering. Video as a result has much more raw data to process than audio. DCTs and quantization are handled effectively using integer operations [12]. Hence fixed-point is preferred over floating point in this case. The conversion IP can be used to convert the floating point data into fixed point for video applications.

The half-precision floating point format is recently introduced in the IEEE 754-2008 standard. This newly introduced format is so far unconsidered but is a very valuable tool of tomorrow's hardware in CPU and GPU. It is represented in a 16-bit format, half that of the traditional single precision. This format is useful for storing floating point numbers as it requires half the storage space and half the memory bandwidth. It ensures better performance in applications that load and store floating point values in certain scenarios where 16-bit precision is sufficient. Since they are half the size, they take up half the cache space, which frees up space for other data in the program. It is also proven that applications like speech recognition, computational fluid dynamics and shock hydrodynamics use more number of bits than required for accuracy and precision, thus the number of bits can be reduced safely [13]. Hence, half-precision format has tremendous application for the future. One application where the half-precision format has been tested is the computed tomography (CT) scan.

<u>CT image reconstruction with half precision floating-point values:</u>

The CT image reconstruction process is a highly computation intensive task. Traditionally the 32 bit single-precision floating point format (float) is used to store and compute for the reconstruction algorithm. Instead of this, if the 16 bit half-precision format is used to store and represent the data in image domain and in the rawdata domain then there is a reduction in the data traffic by 50% on the memory bus. Also, on comparing the float reconstructions and half reconstructions for different images, it is observed that the impact of the quantization noise, which is caused by the reduction in precision, is negligible. Thus half-precision floating point values allow speeding up CT image reconstruction without compromising image quality [8].

The proposed IP is capable of converting from single precision or double precision to half precision floating point format and also vice-versa. Thus it is ideal for the above mentioned CT scan image reconstruction application.

2. Related work

The floating point representations are specified in the IEEE-754 2008 standard [1]. It also includes the conversions between the various formats. Majority of the floating to fixed point conversion algorithms have been implemented in software. Many algorithms have been suggested in [3, 4, 5, 6] for automatic conversion between floating and fixed point formats. These are targeted towards microcontrollers and embedded applications and are not suited for high performance computing. In [9] another approach for conversion is proposed with variable trade-off between computational complexity and accuracy loss, which are the two main factors to consider while converting between floating and fixed point. Certain program translators have been also developed targeted at certain processors like the TMS 320C25, these are proposed in [7]. Thus, majority of the conversions between floating and fixed point has been suggested in software, which is slow and inefficient for large and complex systems. However, in [2] an abstract level hardware implementation of a IP is suggested, which is implemented and tested on a FPGA. This IP includes the conversions between and various floating point formats and also between floating and fixed point formats. However, it does not include the newly introduced half precision floating point format and also the fixed point representation is not uniform which makes it difficult to be used with any particular architecture.

3. Representation of numbers

3.1 Floating point representation

The floating point representation is one way to represent real numbers in a computer. It is quite similar to a scientific notation i.e. it includes a base and an exponent. The reason of using floating point representation is that it covers a larger range of values for a given number of bits and also provides a dynamic precision, which means that the gap between two adjacent numbers is not uniform, it is small for small numbers and it increases as the magnitude of the number increases. Hence it is capable of storing extremely small and large numbers. The IEEE 754-2008 standard provides all the guidelines with respect to the representation and its conversion from one format to the other. The representation of a floating point number consists of three fields – sign, exponent and significand. Considering a radix-2 or binary format the value of the floating point number is given by,

$$(-1)^{\text{sign}} * 2^{\text{exponent}} * \text{significand}$$

The following figure shows the generalized encoding of a floating point representation,

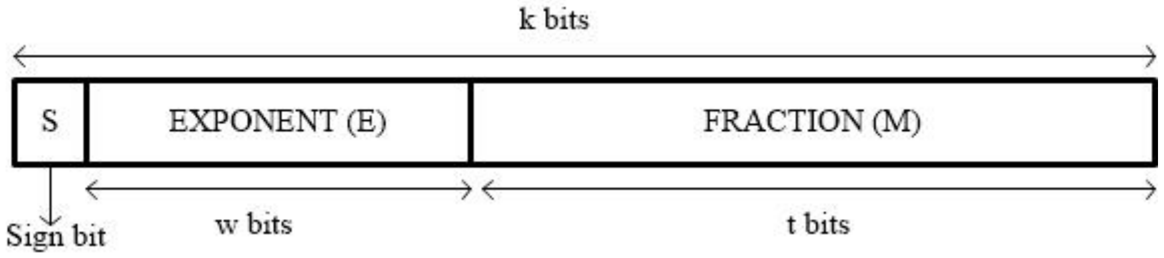

Figure 1.Floating-point generalized representation

The number of bits in each field depends on the type of representation. The two common representations are single precision and double precision. In this work, the newly introduced half precision is also included. The following table lists the three types of floating point formats the number of bits in each of the fields.

Table 1.Number of bits for each field for the floating point formats

Floating point type	Total number of bits (k)	Bits in the exponent field (w)	Bits in the fraction field (t)	Bits in the significand (p)
Single Precision	32	8	23	24
Double Precision	64	11	52	53
Half Precision	16	5	10	11

The sign bit represents whether the number is positive or negative. If the sign is zero, it represents a positive number else if it's a one it represents a negative number. The exponent field is needed to represent both the positive and negative numbers. For every type of representation, there is a fixed bias, which is subtracted from the stored value of exponent in order to get the true exponent. The true exponent is used to calculate the value of the stored number. For example, in case of single precision format, a stored exponent of 200 represents a true exponent of 200 -127 = 73. (127 is the fixed bias used for single precision formats). The fraction field constitutes the fractional bits of the number and along with the implicit leading bit form the significand of the number. The value of the leading bit is generally one, unless the number is a denormal number, which is discussed later.

3.1.1 Normal numbers

Each of the three floating point formats has a different range of numbers and precision that can be represented in that particular format. This is due to the difference in the number of bits in the exponent and fraction fields. The double precision format, having 11 bits and 52 bits in the exponent and fraction fields respectively, has the largest range and maximum precision. On the other hand the half precision format has only 5 bits in the exponent field and 10 bits in the fraction field, hence has the minimum range and precision. The representations where the exponent field (E) is equal to zero or to its maximum possible value (2^w - 1) is reserved for special cases and will be discussed later.

All other cases constitute the normal numbers and can be evaluated as follows,

$$(-1)^{\text{sign}} * 2^{(E - \text{bias})} * 1.M,$$

where M represents the fraction value and the bias is dependent on the type of representation used. The following table lists down the bias and range of the three floating point representations. The range only mentions the positive numbers; however the same is true for negative numbers as well.

Table 2.Range of the floating point representations for normal numbers

Floating point type	**Fixed bias**	**Range of normal numbers (approx.)**
Single precision	127	1.175×10^{-38} to 3.402×10^{38}
Double precision	1023	2.225×10^{-308} to 1.798×10^{308}
Half precision	15	65504 to 6.1035×10^{-5}

3.1.2 Denormal numbers and special cases

As mentioned earlier, the representations where the exponent is equal to zero or its maximum value (2^w- 1) is reserved for special cases. These special cases are discussed in this section,

1. Denormal numbers – When the exponent is zero and the fraction is non-zero, the number is termed as denormal or subnormal number. They are used to represent extremely small numbers. Also, the leading bit of the significand is assumed to be zero unlike normal numbers where it is a one. The exponent value is taken to be the minimum (emin) and it depends on the floating point representation type. The value of the number is given by,

$$(-1)^{\text{sign}} * 2^{\text{emin}} * 0.M$$

The following table represents the value of emin and the range of the denormalized numbers for all the three floating point types. It includes the range for the positive numbers only, but the same is true for negative numbers as well.

Table 3.Range of the floating point representations for denormal number

Floating point type	Emin	Range of denormal numbers (approx.)
Single precision	-126	1.1754×10^{-38} to 1.401×10^{-45}
Double precision	-1022	2.225×10^{-308} to 4.940×10^{-324}
Half precision	-14	6.097×10^{-5} to 5.960×10^{-8}

2. Infinity – The cases where the exponent field consists of all 1's i.e. its maximum possible value (2^w - 1) and the fraction is zero, then the number is classified as infinity. Positive and negative infinity are distinguished with the help of the sign bit. These are useful as they allow operations to continue past the overflow situations.

3. Not a Number (NaN) – The representations where the exponent field consists of all 1's i.e. the maximum possible value (2^w - 1) and the fraction is non-zero, then the number is classified as NaN. Positive and negative NaN is distinguished by the sign bit. These are used if the result of an operation is invalid.

4. Zero – When the exponent and fraction are both zero, the value of the number is zero. In floating point representations there are two ways a zero can be represented, i.e. a positive zero and a negative zero. This is determined by the sign it, however, they both are considered equal.

3.2 Fixed point representation

The fixed point representation resembles representation in a simple binary format. The only difference being the presence of an implicit decimal point, which is also called the binary point considering the radix-2 representation. The fixed-point data is made up of an integer part and a fractional part as presented in the following figure. The fixed point representation is defined by three fields, *b*, *m* and *n*, where *b* refers to the total data word length. The terms *m* and *n* define the position of the binary point, *m* refers to the number of bits before the decimal point and *n* refers to the number of bits after the decimal point. In fixed point arithmetic, *m* and *n* remain fixed which results in uniform representation during all computations and data manipulation.

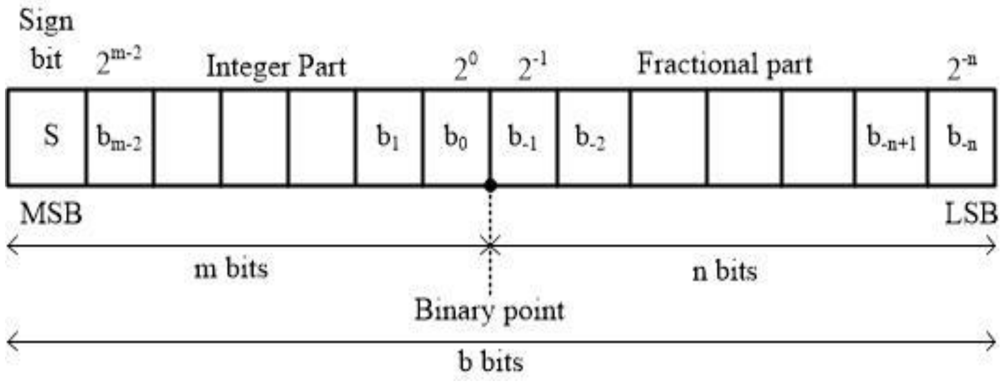

Figure 2.Fixed-point generalized representation

The binary point represents the coefficient of the term $2^0 = 1$. All digits to the left of the binary point carry a weight of 2^1, 2^2, 2^3, and so on, while the digits on the right of the binary point carry a weight of 2^{-1}, 2^{-2} , 2^{-3}, and so on. These weights and the corresponding bits are used to evaluate the value of the fixed-point representation. The negative numbers can be represented using the two’s complement technique. The fixed point representation is extremely simple and can reuse the basic integer arithmetic hardware to perform its functions.

Many standards have been used to specify the fixed-point representation. One such standard is the Q notation. The Q notation is defined as Q*m.n* where *m* represents the bits before the binary point and *n* represents the number of bits after the binary point. For example Q1.14 refers to one integer bit and 14 fractional bits. In this research work, the Q17.15 notation is used, which implies 17 integer bits and 15 fractional bits, and the negative numbers are represented in the two’s complement notation. The range and precision of Q*m.n* notation can be specified as,

Range: $[-2^{-m}, 2^m – 2^{-n}]$

Precision: 2^{-n}

For this research work a 32-bit fixed point representation is used in the Q17.15 format and for negative numbers the two's complement technique is used. The following table specifies the range and precision of the fixed-point format used in this thesis.

Table 4.Range and precision of the Q17.15 fixed point notation

Fixed-point notation	**Range**	**Precision**
Q17.15	-65536 to 65535.999969482	0.000030517578125

3.3 Rounding modes

Conversions between the various floating point formats and from floating to fixed point formats are often accompanied by rounding off the final result. This is necessary because not all the formats can represent all the possible real values; hence we always need to round the result in the desired representation. The IEEE 754-2008 standard specifies four rounding modes which are also used in this research work. They are explained as follows,

1. Round ties to even – In this rounding mode the result nearest to the infinitely precise result shall be delivered; if the two nearest representations are equally near, the one with the even least significant bit must be delivered.

2. Round towards positive infinity – In this rounding mode the result should be the closest to and no less than the infinitely precise result.

3. Round towards negative infinity – In this rounding mode the result should be the closest to and no greater than the infinitely precise result.

4. Round towards zero – In this rounding mode, the result should be the closest to and no greater in magnitude than the infinitely precise result. This is simple truncation.

All these four rounding modes are available in the designed IP and depending on the user input the desired rounding operation is performed. The rounding mode is specified using a 2-bit encoding scheme. The following table specifies the 2-bit code and the respective rounding mode performed.

Table 5.Rounding modes and the corresponding 2 bit code

2-bit code	Rounding operation performed
00	Round ties to even
01	Round towards zero
10	Round towards positive infinity
11	Round towards negative infinity

3.4 Exception signals

These signals are used to indicate any exceptions that arise during any computations or as in this case, conversions from one format to the other. The IEEE 754-2008 standard specifies four exception signals which are also incorporated in the design of the IP. The handling of these exceptions is also provided in the standard. These exception signals are as follows,

1. Invalid – This exception is signaled if and only if there is no definably useful result. In this case the operands are invalid for the operation to be performed. In this research work an invalid exception is raised if the floating point operands are NaN or infinity.

2. Overflow – This exception is signaled if the result obtained exceeds the magnitude that can be represented in the destination format. This exception is common while converting from floating point to fixed point, as the range of floating point is larger than fixed point. This exception is handled, by signaling an overflow flag and the output is the maximum representable number of the destination format.

3. Underflow – This exception is signaled if the result obtained is smaller than the magnitude that can be represented in the destination format. This is also a common exception as the floating point numbers are capable of representing extremely small numbers (denormal), which cannot be expressed in fixed point format.

4. Inexact – This exception is raised whenever a rounding operation is performed, i.e. the result differs from what would have been computed if the range and precision were unbounded.

All the above exception signals act as output signals of the proposed IP and depending on the operands and operation performed they are signaled accordingly.

4. Design of the proposed IP

The following figure shows the top level block diagram of the proposed conversion IP.

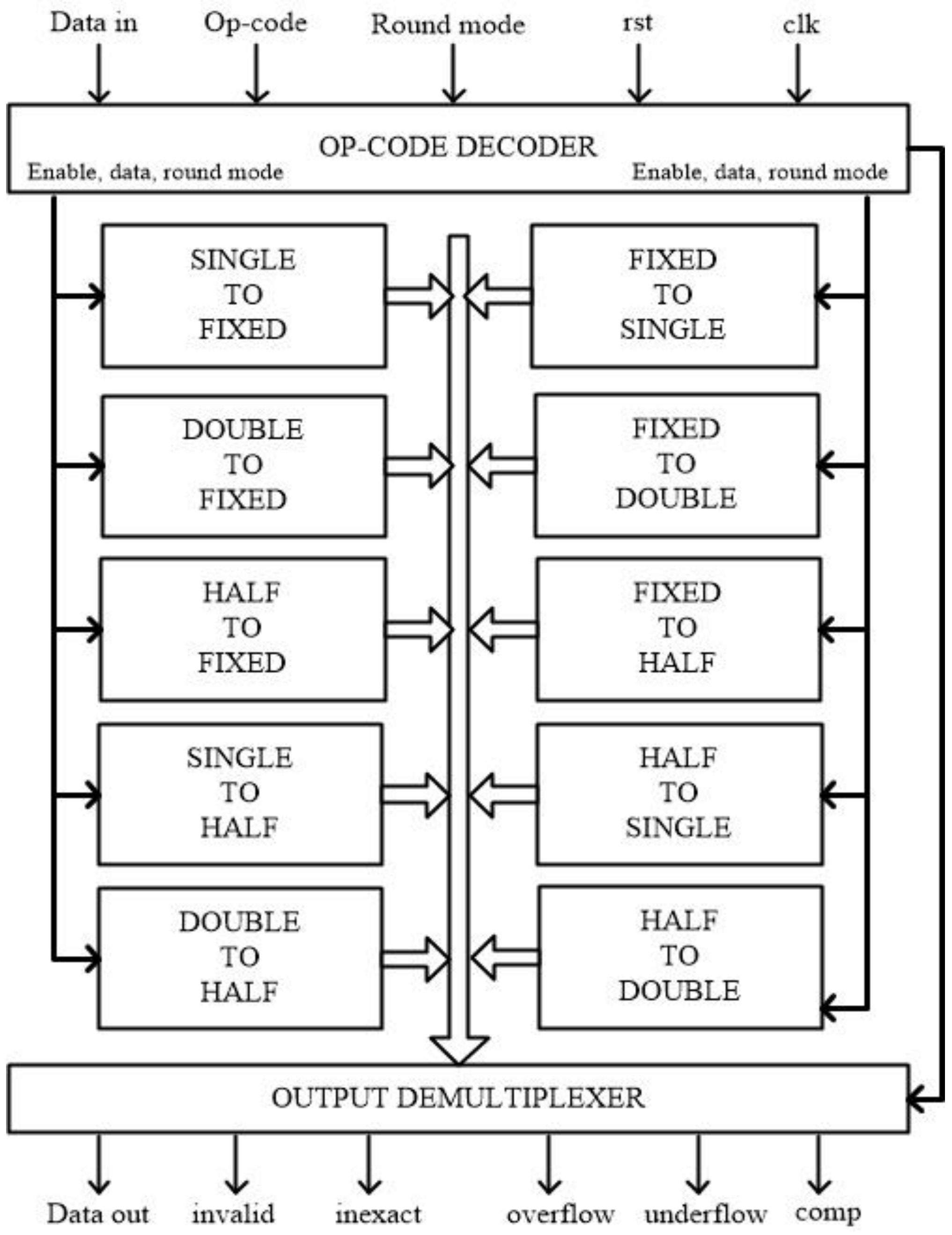

Figure 3.Top-level block diagram of the conversion IP [2]

The IP is capable of performing ten operations depending on the op-code. The op-code is encoded using 4 bits. The operations include the conversion from floating point formats (half, single and double) to fixed point format and vice-versa. Also, given the advantages of the recently introduced half-precision floating point format, the IP also includes conversions from single and double precision to half precision and vice-versa. The following table lists the op-codes and the corresponding operation performed.

Table 6.Op-code and the corresponding operation

Op-code (4 bits)	Operation performed
0001	Single precision to Fixed-point
0010	Double precision to Fixed-point
0011	Half precision to Fixed-point
0100	Fixed-point to Half precision
0101	Fixed-point to Double precision
0110	Fixed-point to Single precision
0111	Single precision to Half precision
1000	Double precision to Half precision
1001	Half precision to Double precision
1010	Half precision to Single precision

The conversion IP accepts the input data along with a 2-bit rounding mode and the op-code. The input decoder decodes the op-code and enables the respective module required for the conversion. The input data and the rounding mode act as the input to the module and the conversion operation is performed. After the conversion is complete, the result is forwarded to the main data bus, and depending on the op-code, the output de-multiplexer takes the output data and the exception signals from the data bus and generates the result. The five exception signals are also generated depending on the conversion. Another signal, *comp* is used to indicate the completion of the conversion and when it is high it indicates that the output is valid, and when it is low, it means that the conversion is still in process and the output is invalid. All the individual modules are designed using 5-stage pipeline architecture and along with the op-code decoder and the output de-multiplexer, the conversion IP effectively acts as 7-stage pipeline architecture. The following table lists down all the input-output pins along with the width and their function,

Table 7.I/O pins description

I/O pin	Direction	Width	Function
Data in	Input	64	Input data
Op-code	Input	4	Specifies the conversion to be performed.
Round mode	Input	2	Specifies the desired rounding mode.
rst	Input	1	Resets the IP.
clk	Input	1	System clock.
Data out	Output	64	Output data in the destination format.
Invalid	Output	1	Exception signals
Inexact	Output	1	
Overflow	Output	1	
Underflow	Output	1	
Comp	Output	1	Indicates completion of the conversion operation

The following section focuses on the individual modules and their description using the SystemVerilog language.

4.1 Floating-point to fixed-point format conversion

This section discusses the algorithm used to convert the floating point formats i.e. half, single and double precision into fixed point format and also how they are described using SystemVerilog. The input to these modules is the data in the half, single or double precision format and is converted into the Q17.15 fixed point format. The algorithm to convert from floating point to fixed point can be described in the following steps,

1. The floating point number is first classified, whether it is a normal number, denormal number, NaN, infinity or zero. If the number is NaN or infinity, the conversion cannot be performed and the invalid exception signal is raised. Also, the sign, exponent and fraction of the number are separated.

2. The exponent of the number is used to check whether the number can be represented in the Q17.15 format. If the number is beyond the range of numbers that can be represented by the Q17.15 format, then the overflow or underflow exception is raised accordingly.

3. The true exponent value is calculated by subtracting the fixed bias from the exponent value. The fixed bias depends on the floating point format. This difference is equal to the number of shifts required to convert it into the fixed-point format.

4. The implicit leading bit of the significand is appended to the fraction and the shifting operation is performed. The integer and fractional parts of the fixed point representation are separated from the fraction based on the shift amount.

5. After the shifting, the rounding operation is performed and the result is generated in the Q17.15 format. The result is represented in the two’s complement form.

The algorithm can be summarized using the following flow-chart,

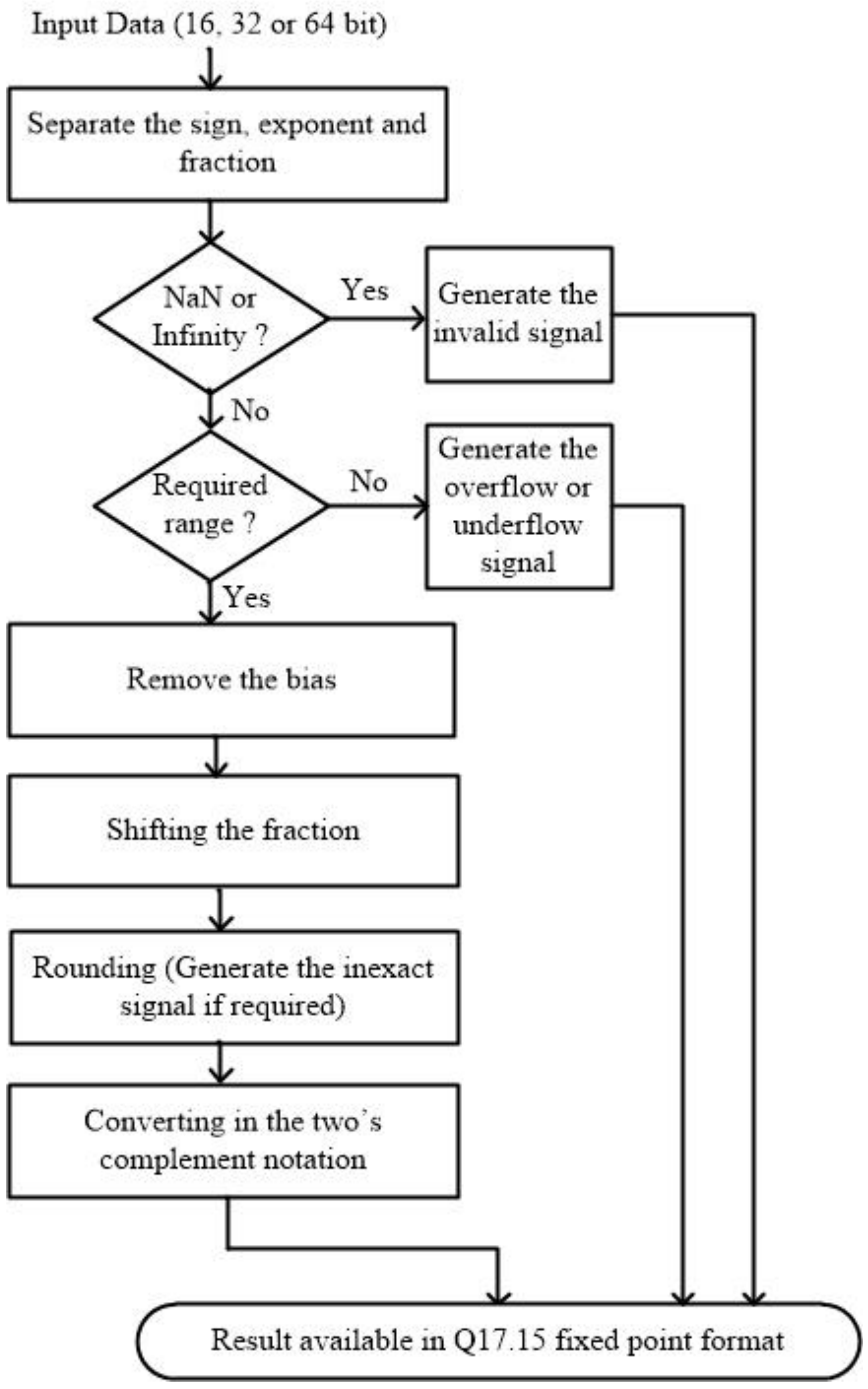
Input Data (16, 32 or 64 bit)
Separate the sign, exponent and fraction
NaN or Infinity ?
Yes
Generate the invalid signal
No
Required range ?
No
Generate the overflow or underflow signal
Yes
Remove the bias
Shifting the fraction
Rounding (Generate the inexact signal if required)
Converting in the two's complement notation
Result available in Q17.15 fixed point format

Figure 4.Flow-graph of the algorithm used to convert floating point to fixed point

The above algorithm is used to design the three modules, namely, single precision to fixed-point, double precision to fixed-point and half precision to fixed-point. Each module is designed as a 5 stage pipeline. The generalized block diagram of each module is shown in the following figure,

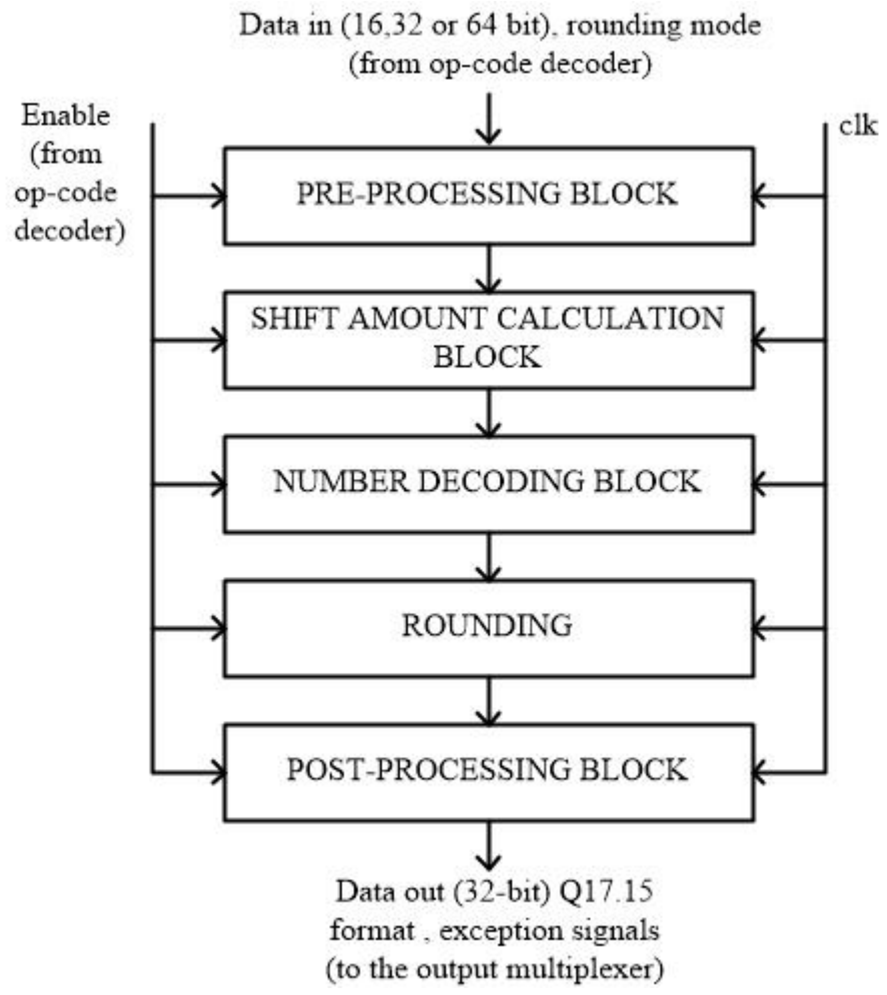
Data in (16,32 or 64 bit), rounding mode
(from op-code decoder)
Enable (from op-code decoder)
clk
PRE-PROCESSING BLOCK
SHIFT AMOUNT CALCULATION BLOCK
NUMBER DECODING BLOCK
ROUNDING
POST-PROCESSING BLOCK
Data out (32-bit) Q17.15 format , exception signals (to the output multiplexer)

Figure 5.Floating point to fixed point module block diagram [2]

Each of the above blocks is implemented as having a combinational block and a set of flip-flops with would act as the pipeline registers. The function performed in each block is as follows,

1. Pre-processing block: This block is responsible for separating the sign, exponent and fraction of the input floating point number. After this it also uses the exponent to classify, whether the number is NaN, infinity, denormal, or zero.

2. Shift amount calculation block: This block generates the exception signals. If the number is NaN or infinity the invalid signal is generated. If the number is beyond the range that can be represented by the fixed point format, then the overflow or underflow signal is generated. This block also appends the implicit leading bit of the significand to the fraction and removes the bias from the exponent value.

The following table shows the fixed bias for each of the floating point formats as well as the range of numbers that can be represented in the Q17.15 format based on the exponent value.

Table 8.Bias and range of the numbers representable in the Q17.15 format

Floating point type	**Fixed Bias**	**Range of numbers**	
		Emax	**Emin**
Single Precision	127	142	112
Double Precision	1023	1038	1008
Half Precision	15	30	0 (All denormal numbers not included)

3. Number decoding block: This block is responsible for the shifting operation and separating the integer and fractional part of the fixed point format from the fraction. It also prepares the number for rounding operation by storing three special bits, guard, round and the sticky bit. These three bits are used to determine whether rounding is needed or not. These bits are derived from the section of bits which cannot be accommodated in the required 32 bits. This can be explained in the following figure,

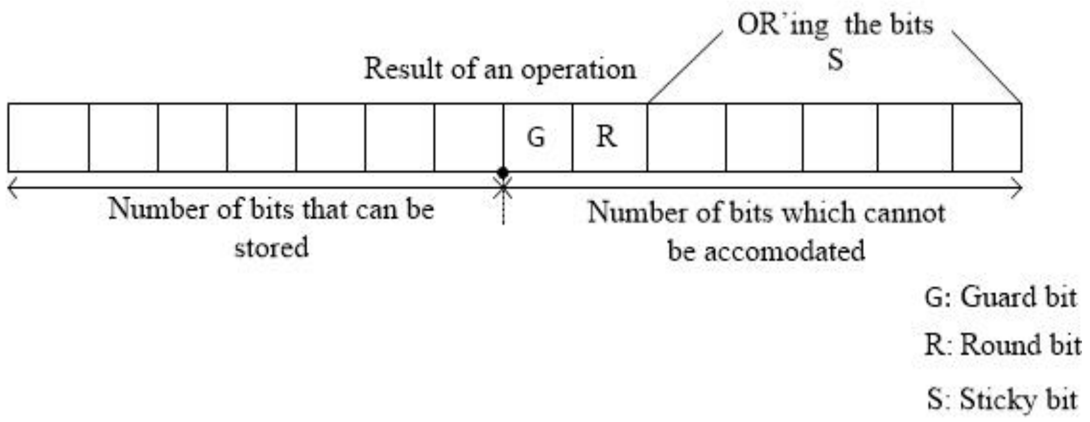

Figure 6.Rounding bits

4. Rounding block: This block performs the rounding operation based on the rounding mode and the guard, round and sticky bits. Also, if any rounding is performed the inexact exception signal is generated. As mentioned earlier, there are four rounding modes supported in this design. If *rnd* is the bit to be added to the result for rounding, *rnd* can be determined based on the following table,

Table 9.Rounding operation

2-bit encoding	Rounding mode performed	Value of *rnd*
00	Round ties to even	*rnd* = guard_bit & (LSB | round_bit | sticky_bit)
01	Round towards zero	*rnd* = 0
10	Round towards positive infinity	*rnd* = (~sign) & (guard_bit | round_bit | sticky_bit)
11	Round towards negative infinity	*rnd* = sign & (guard_bit | round_bit | sticky_bit)

5. Post-processing block: This block generates the result in the Q17.15 format and in the two’s complement notation.

The combinational and pipeline registers are implemented using the `always_comb` and `always_ff` blocks of SystemVerilog. The `always_comb` block implies that the logic within it is combinational in nature and `always_ff` block implies that the logic within it is sequential. This is an improvement over the traditional Verilog language, where the sensitivity list would determine the nature of logic and would lead to ambiguous results on synthesis. An example of the `always_comb` and `always_ff` block is shown below,

```
//This is a piece of code from the pre-processing block of
single-precision floating point to fixed point conversion.
always_comb begin
      sign_int = single_data_in[31];
      exponent_int = single_data_in[30:23];
      mantissa_int = single_data_in[22:0];

      if ((exponent_int == 0) && (mantissa_int != 0))
           denormal_int = 1'b1;
      else
           denormal_int = 1'b0;

      if ((exponent_int == 255) && (mantissa_int != 0))
           NaN_int = 1'b1;
      else
           NaN_int = 1'b0;

      if ((exponent_int == 255) && (mantissa_int == 0))
           infinity_int = 1'b1;
      else
           infinity_int = 1'b0;

      if ((exponent_int == 0) && (mantissa_int == 0))
           zero_int = 1'b1;
      else
           zero_int = 1'b0;
   end

   always_ff @(posedge clk) begin
      if (enable == 1) begin
           r_mode <= round_mode;
```

```
            sign <= sign_int;
            exponent <= exponent_int;
            mantissa <= mantissa_int;
            denormal <= denormal_int;
            NaN <= NaN_int;
            infinity <= infinity_int;
            zero <= zero_int;
       end
       else begin
            r_mode <= 0;
            sign <= 0;
            exponent <= 0;
            mantissa <= 0;
            denormal <= 0;
            NaN <= 0;
            infinity <= 0;
            zero <= 0;
       end
    end
```

4.2 Fixed-point to floating point format conversion

This section covers the algorithm used to convert the fixed point format (Q17.15) into the various floating point formats. The input to these modules is the fixed-point representation and depending on the op-code the input is converted to one of the floating point formats-half, single or double precision. The algorithm used to convert from fixed point to floating point can be stated as follows,

1. The input data is first converted from its two's complement notation to the normal representation. This gives the sign of the input data.

2. Identify the position of the leading one in the representation. This is obtained by shifting the number to the left till the most significand bit is a one.

3. The position of the leading one gives the true exponent value, which needs to be added to the fixed bias of the floating point format to get the biased exponent value.

4. The shifted number represents the fraction of the floating point format, with the leading one dropped. The leading one is always implied to be stored in the floating point format. The size of the stored fraction depends on the floating-point format (23 for single precision, 52 for double precision and 10 for half precision).

5. The fraction is then rounded based on the rounding mode specified by the user. If required, the inexact exception signal is generated. The sign, exponent and fraction are then concatenated to get the desired floating point format.

The overflow, underflow and invalid exception signals are not generated in this case. This is because the range of single and double precision floating point numbers is much larger than the fixed point numbers hence no cases of overflow and underflow occur. However, for half precision numbers, there might be a case of overflow as the maximum number that can be represented in the Q17.15 fixed point format, exceeds the maximum representable number in the half precision format. This is shown in the following table,

Table 10.Range of half precision vs. Q17.15 fixed point

Range of Half precision number	**Range of Q17.15 fixed point**
65504 to 5.960×10^{-8}	-65536 to 65535.999969482

The algorithm can be represented by the following flow-graph,

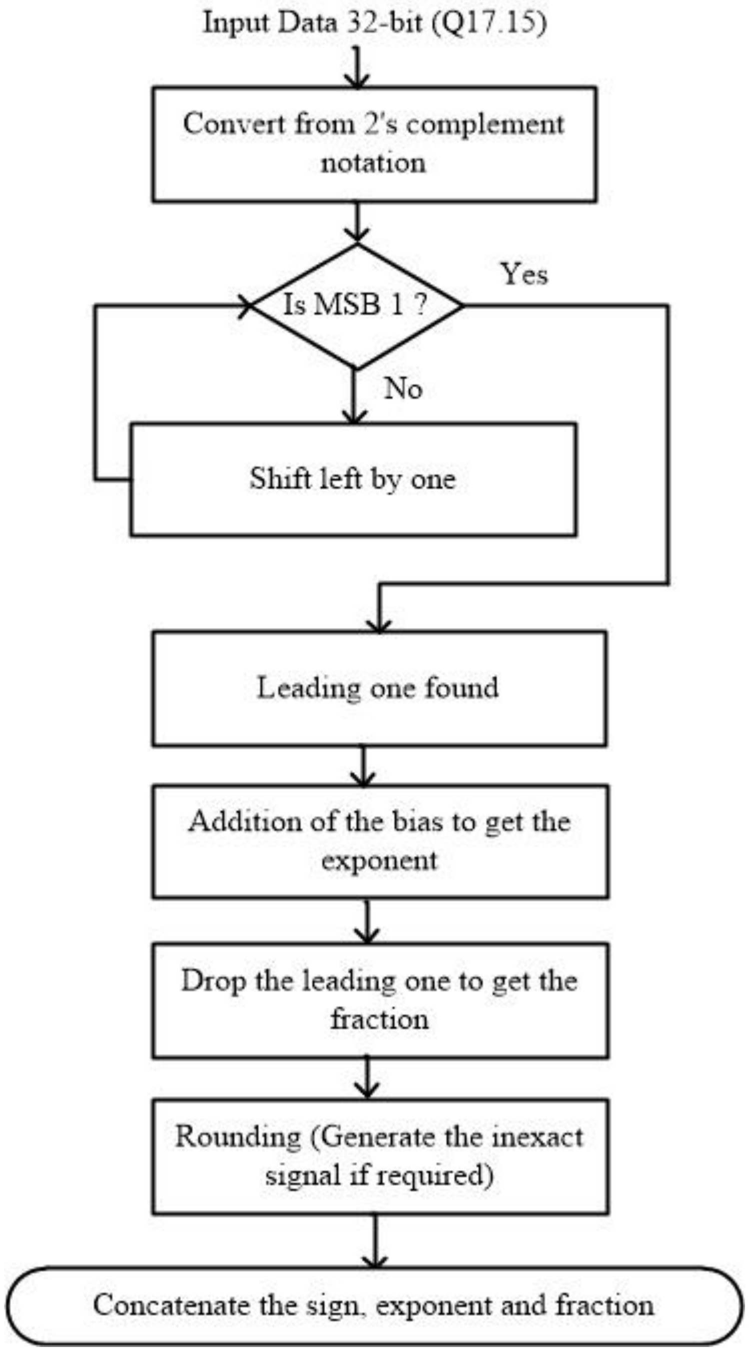
Input Data 32-bit (Q17.15)
Convert from 2's complement notation
Is MSB 1 ?
Yes
No
Shift left by one
Leading one found
Addition of the bias to get the exponent
Drop the leading one to get the fraction
Rounding (Generate the inexact signal if required)
Concatenate the sign, exponent and fraction

Figure 7.Flow-graph of the algorithm used to convert fixed point to floating point

The above algorithm is used to design the three modules, namely, fixed point to single precision, fixed-point to double precision and fixed-point to half precision. Each module is designed as a 5 stage pipeline. The generalized block diagram of each module is shown in the following figure,

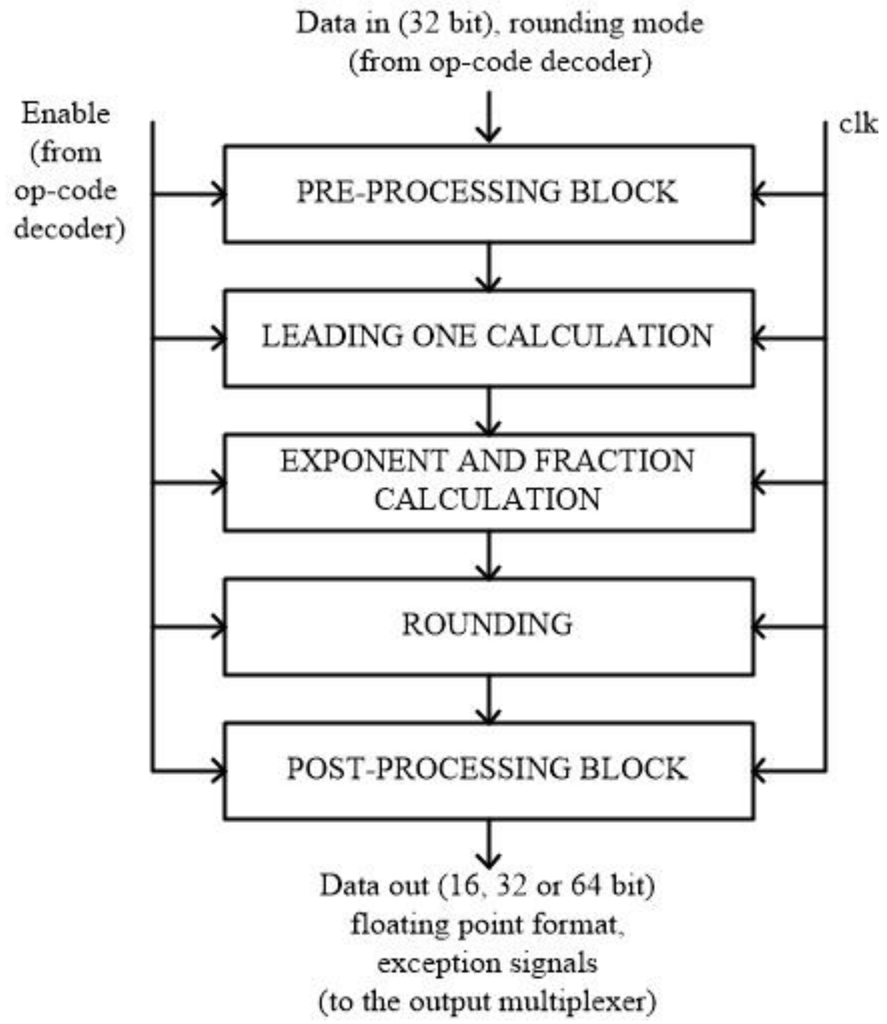

Figure 8.Fixed point to floating point module block diagram [2]

Each block is implemented as a combinational and sequential block. The sequential block acts as the pipeline registers. The function of each block is as follows,

1. Pre-processing block: This block is responsible for converting the number from two’s complement notation to the absolute representation. This block also stores the sign of the number.

2. Leading one calculation: This block identifies the leading one by shifting the number to the left. As a result it stores the position of the leading one in the number and forwards the shifted result to the next block.

3. Exponent and fraction calculation: Here, the position of the leading one is added to the fixed bias of the floating point format desired to get the biased exponent. Also, from the shifted result, the leading one is dropped to get the fraction. The width of the fraction depends on the type of floating point format. Also, the guard, round and sticky bits are stored which are needed for the rounding operation.

4. Rounding: In this block, the rounding operation is performed depending on the rounding mode specified by the user and the guard, round and sticky bits. The inexact exception signal is generated if required.

5. Post-processing: The sign, exponent and the rounded fraction are concatenated to get the result in the floating point format.

Similar to the floating to fixed point modules, these modules are also implemented as a 5-stage pipeline using the `always_comb` and `always_ff` blocks of System Verilog.

4.3 Single and double precision to half precision floating point conversion

The newly introduced half-precision format holds very high promise for many applications where floating point numbers are used but do not require the large range and precision offered by the traditional single and double precision format. Since half precision format is made up of 16 bits, it occupies half the space occupied by single precision format, and thus saves memory and it also efficient while loading and storing data from cache. Thus, the IP is also designed to support conversion between the single/double precision and half precision formats. This section discusses the algorithm used to convert from single and double precision to half precision format. The algorithm used to convert from single/double precision to half precision format is as follows,

1. First the sign, exponent and fraction of the input data need to be separated out. The number is classified as NaN or infinity. If the number is NaN or infinity, the operation cannot be performed and the invalid exception signal is raised. Also, the overflow and underflow exception signals are raised if the input is beyond the range of the numbers that can be represented in half precision format.

2. To calculate the exponent, the fixed bias of the input floating point format (127 for single precision and 1023 for double precision) is subtracted from the input exponent value. Also, the fixed bias of the half precision i.e.15 needs to be added to get the biased exponent value.

3. If the resultant exponent is equal to or less than zero, it implies that the number in the output half precision format will be a denormal number.

4. If the number is denormal, a leading one needs to be appended to the input fraction and it is shifted to the right by the difference obtained in step 2. However, if the number is normal no shifting is required.

5. The first ten bits of the fraction obtained after step 4 form the fraction of the half precision format. Rounding operation is performed depending on the guard, round and sticky bits. The sign, exponent and fraction are then concatenated to get the resultant half precision format.

The above algorithm can be explained using the following flow-graph,

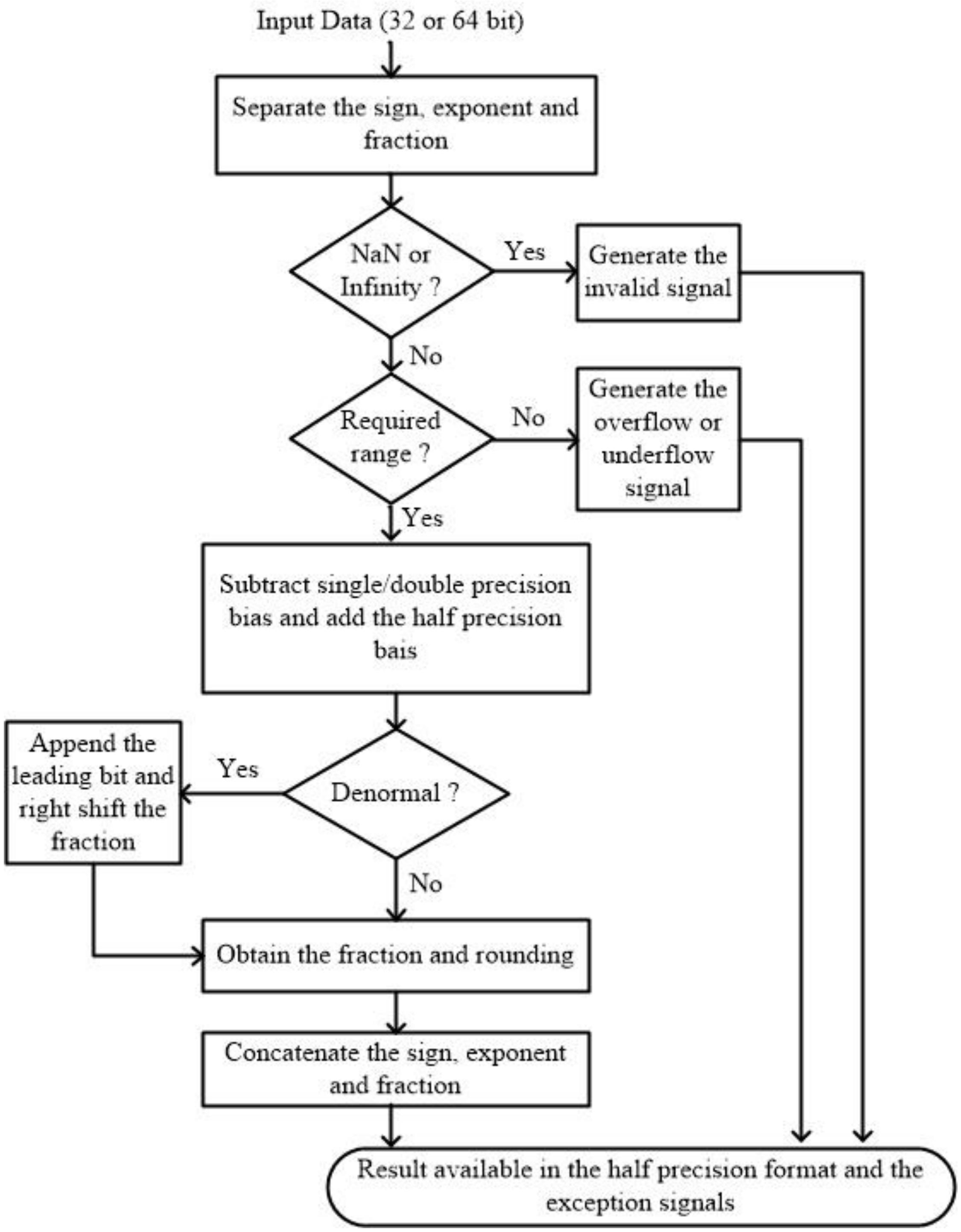
Input Data (32 or 64 bit)
Separate the sign, exponent and fraction
NaN or Infinity ?
Yes
Generate the invalid signal
No
Required range ?
No
Generate the overflow or underflow signal
Yes
Subtract single/double precision bias and add the half precision bais
Denormal ?
Yes
Append the leading bit and right shift the fraction
No
Obtain the fraction and rounding
Concatenate the sign, exponent and fraction
Result available in the half precision format and the exception signals

Figure 9.Flow-graph of the algorithm used to convert single/double precision to half precision floating point format

The above algorithm is used in two modules i.e. single precision to half precision and double precision to half precision. The block diagram of these two modules is as shown below,

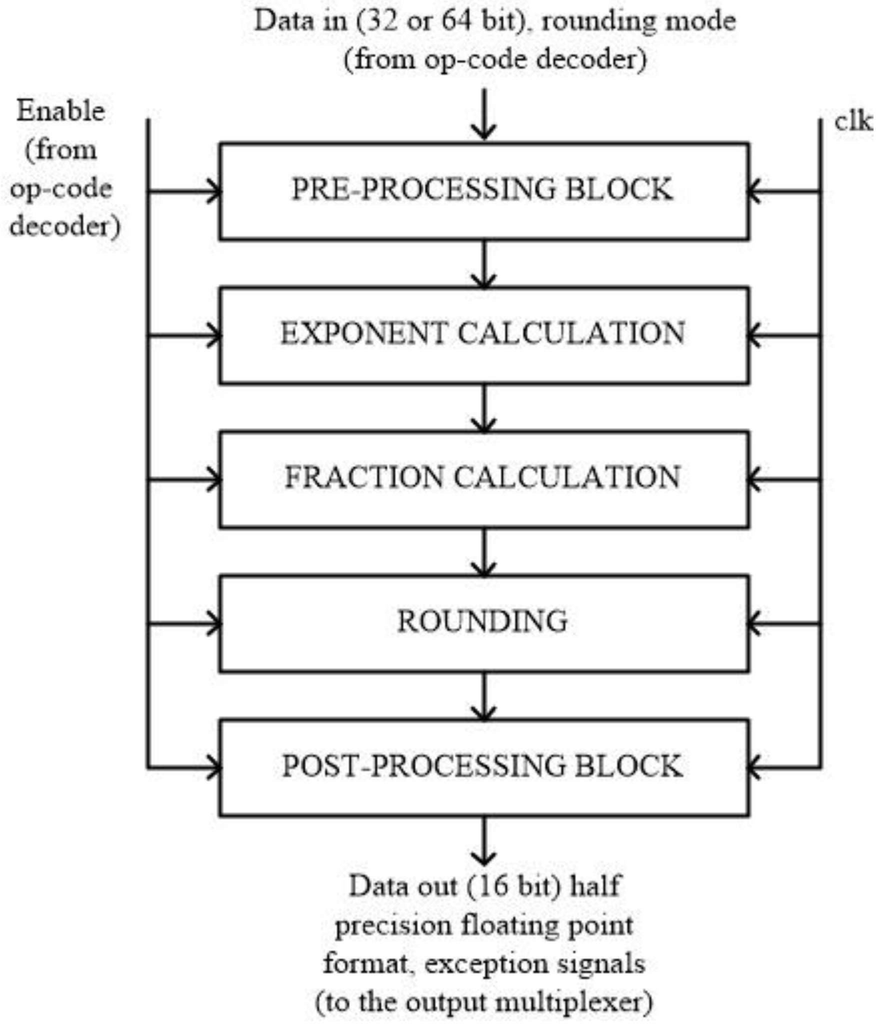
Data in (32 or 64 bit), rounding mode (from op-code decoder)
Enable (from op-code decoder)
clk
PRE-PROCESSING BLOCK
EXPONENT CALCULATION
FRACTION CALCULATION
ROUNDING
POST-PROCESSING BLOCK
Data out (16 bit) half precision floating point format, exception signals (to the output multiplexer)

Figure 10.Single/Double precision to half precision module block diagram

The function of each block is described below,

1. Pre-processing block: This block is used to separate the sign, exponent and fraction of the input single or double precision format. It also classifies the number as NaN or infinity and raises the invalid exception. It also uses the exponent value to indicate whether the output half precision number will be denormal or normal. It also raises the

overflow or underflow exceptions. The following table lists the ranges of the exponent values of the single and double precision for which the exceptions will be raised and also when the output number would be denormal.

Table 11.Range of exponent values casing overflow, underflow and denormal cases

Floating-point type	**Exponent value causing overflow**	**Exponent values causing underflow**	**Exponent values for which output will be denormal**
Single Precision	>142	<103	112 to 103
Double Precision	>1038	<999	999 to 1008

2. Exponent calculation stage: This block is responsible for removing the single or double fixed bias and adding the half precision bias to get the exponent value. If the output number is denormal the exponent is equal to zero, and the difference obtained by the above calculation is stored.

3. Fraction calculation stage: This block checks whether the output number is a denormal number, if it is then the leading one is appended to the input fraction at the most significant position and it is shifted to the right by the difference obtained in the previous block. However, if the number is normal, no such shifting is required. The first ten bits act as the fraction of the half precision format. Also, the guard, round and sticky bits are stored for rounding purposes.

4. Rounding stage: In this block, the fraction is rounded based on the guard, round and sticky bits and the input rounding mode. Also, the inexact exception is raised if applicable.

5. Post-processing stage: Here, the sign, exponent and the fraction bits are concatenated to obtain the output floating point number in half-precision format.

4.4 Half precision to Single and double precision floating point conversion

This section covers the conversion from half precision to single or double precision floating point. The algorithm is as follows,

1. The sign, exponent and fraction of the half precision format need to be separated out. The number is classified as NaN, infinity or denormal. If the number is NaN or infinity, the invalid signal is raised and the conversion operation cannot be carried out.

2. If the number is denormal, the position of the leading one of the mantissa is calculated. This can be achieved by shifting the fraction to the left.

3. If the number is denormal, the fraction is shifted to the left till the most significant bit is a one. This bit is dropped and the resultant number acts as the fraction for the single or double precision. However, if the number is not denormal, no shifting is required and the input fraction is the output fraction of the single or double precision with appended zeroes.

4. The exponent is calculated by first subtracting the half precision bias (15) from the input exponent and then adding the single or double precision bias (127 or 1023). If the number is denormal, then even the leading one position calculated in step 2 needs to be subtracted in order to get the exponent value.

5. The sign, exponent and fraction are concatenated to get the single or double precision floating point format.

No rounding is required in this conversion as the range and precision of single and double precision format is much more than half precision floating point format. Thus, all the numbers that are representable in half precision format can be represented in single and double precision format. The above algorithm can be explained in the following flow-graph,

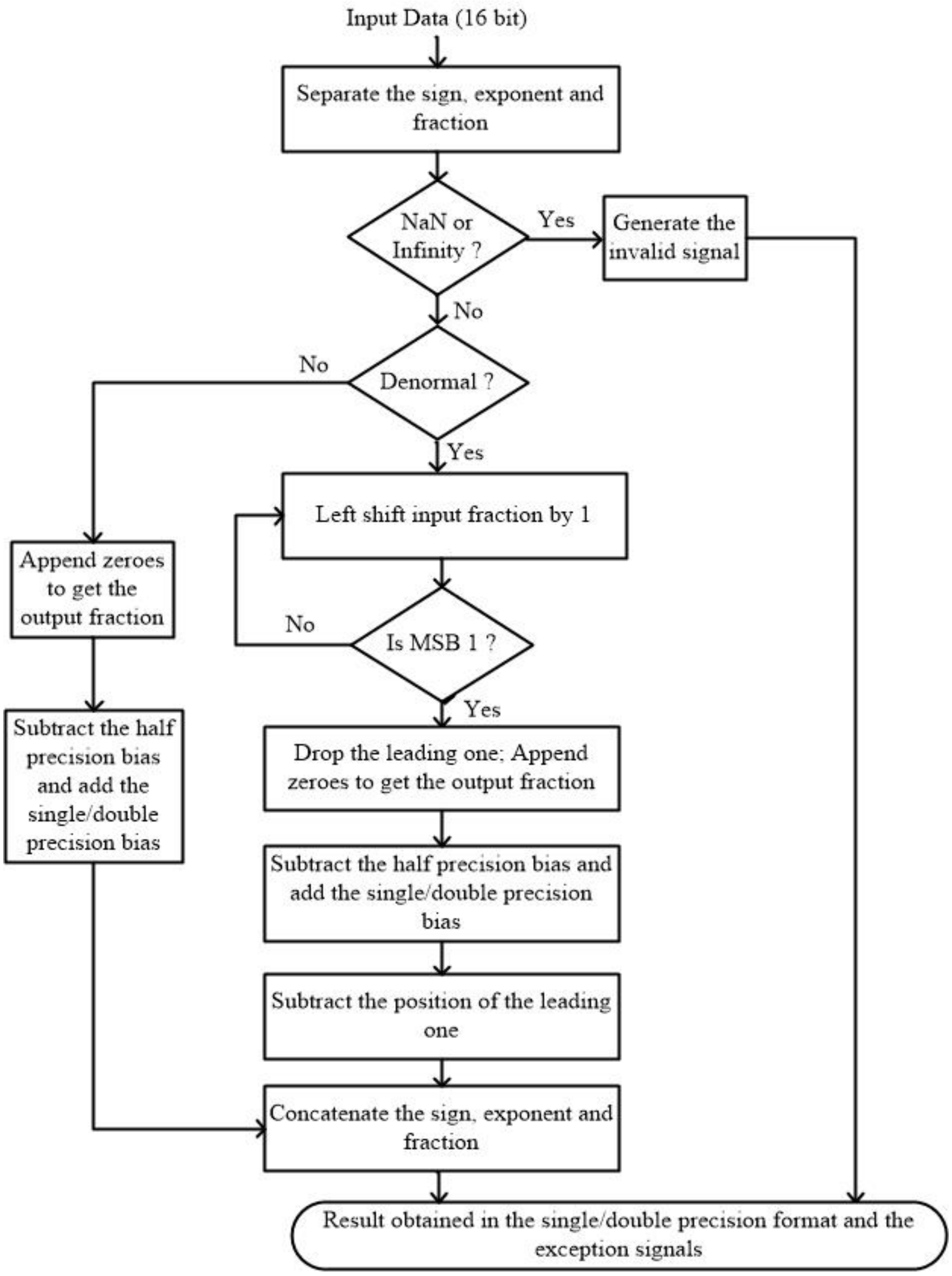
Input Data (16 bit)
Separate the sign, exponent and fraction
NaN or Infinity ?
Yes
Generate the invalid signal
No
No
Denormal ?
Yes
Left shift input fraction by 1
Append zeroes to get the output fraction
No
Is MSB 1 ?
Yes
Subtract the half precision bias and add the single/double precision bias
Drop the leading one; Append zeroes to get the output fraction
Subtract the half precision bias and add the single/double precision bias
Subtract the position of the leading one
Concatenate the sign, exponent and fraction
Result obtained in the single/double precision format and the exception signals

Figure 11.Flow-graph of the algorithm used to convert half precision to single/double precision floating point format

The above algorithm is implemented in two modules, i.e. half precision to single precision floating point and half precision to double precision floating point conversion. The top-level block diagram of these modules is as shown,

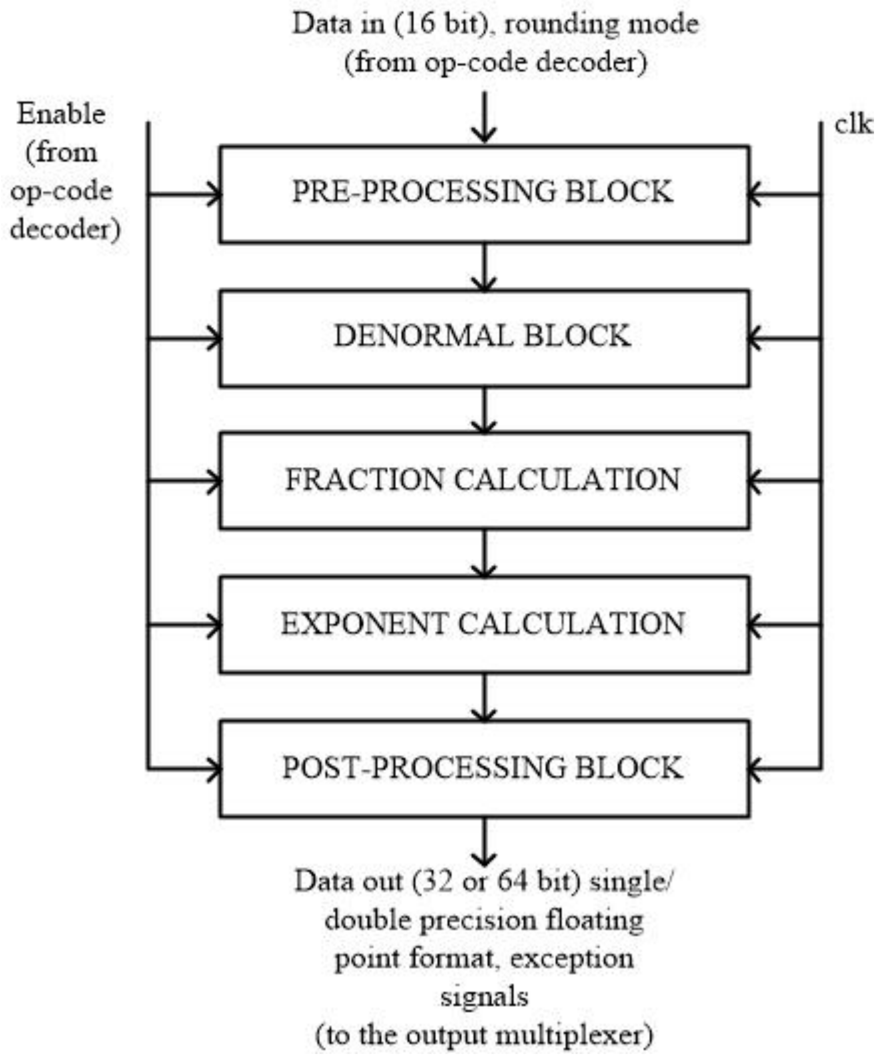
Data in (16 bit), rounding mode (from op-code decoder)
Enable (from op-code decoder)
clk
PRE-PROCESSING BLOCK
DENORMAL BLOCK
FRACTION CALCULATION
EXPONENT CALCULATION
POST-PROCESSING BLOCK
Data out (32 or 64 bit) single/ double precision floating point format, exception signals (to the output multiplexer)

Figure 12.Half precision to single/double precision module block diagram

The functions of each block is as follows,

1. Pre-processing block: This block separates the sign, exponent and fraction from the input half precision floating point format. It also classifies the numbers as NaN, infinity or denormal.

2. Denormal block: If the number is denormal, this block calculates the position of the leading one in the fraction by shifting it to the left and checking the most significant bit. This is useful to calculate the exponent and output fraction. If the number is not denormal, no shifting is necessary and the number is forwarded to the next stage.

3. Fraction calculation block: The shifted fraction from the previous stage acts as the input to this block. The leading one bit is dropped and the resultant is appended with zeroes to get the output fraction. If the number was not denormal, the input from the previous stage is directly appended with zeroes.

4. Exponent calculation block: This block, removes the half precision bias and adds the single or double precision bias to the input exponent value. If the input number is denormal, as determined by the pre-processing block, then the position of the leading one determines the exponent value.

5. Post-processing block: Here, the sign, exponent and fraction are concatenated to get the final result in the single or double precision format.

The top level design is implemented by instantiating the ten conversion modules along with the op-code decoder and output de-multiplexer. The next section focuses on the compilation and simulation results of each module as well as the top level design of the IP.

5. Compilation and Simulation results

Each of the individual modules was compiled using Synopsys VCS. In-order to test each module, a testbench was developed, which would generate the signals acting as inputs to the modules. The following sections focus on the testbench and the simulation results for each module,

Initially to check the functionality of the modules, directed testbenches were developed, i.e. the input vectors were manually written and the results were observed on the waveforms to check for correctness. The waveforms of these tests are shown below. In the waveforms, we can observe that the output result and the exception signals are available after 5 clock cycles as the design is a 5 stage pipeline.

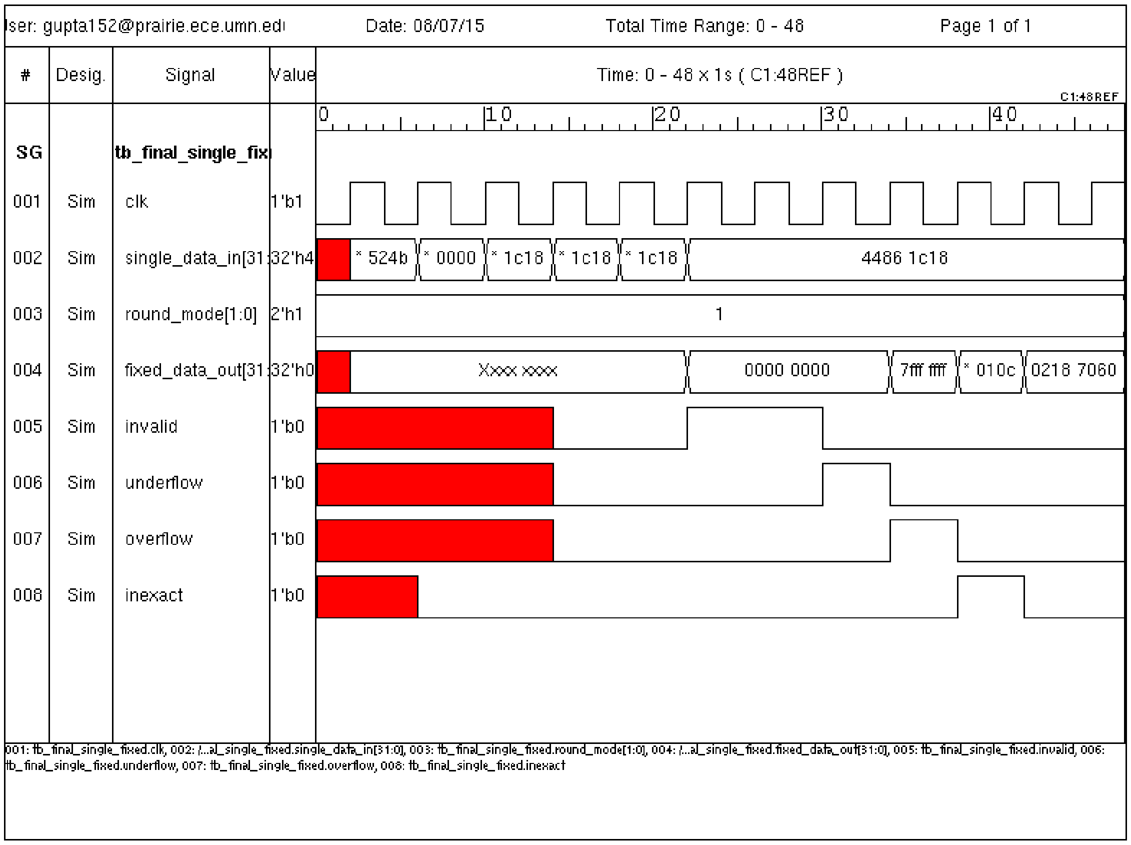

Figure 13.Waveform of single precision to fixed-point conversion

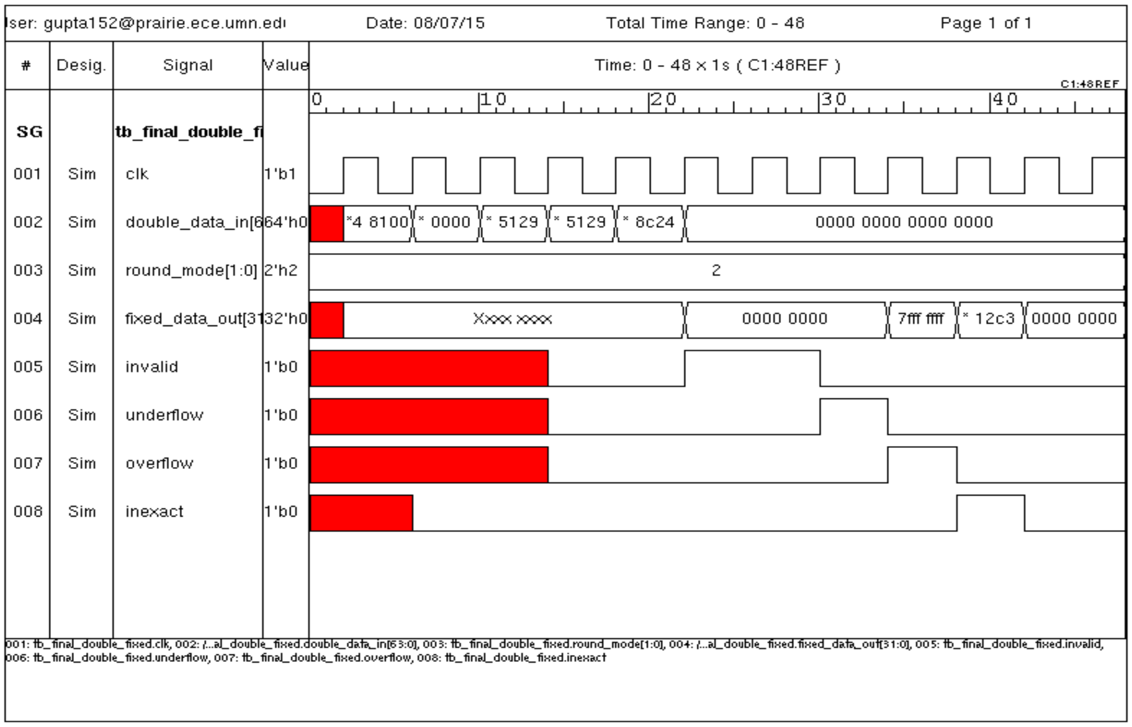

Figure 14.Waveform of double precision to fixed-point conversion

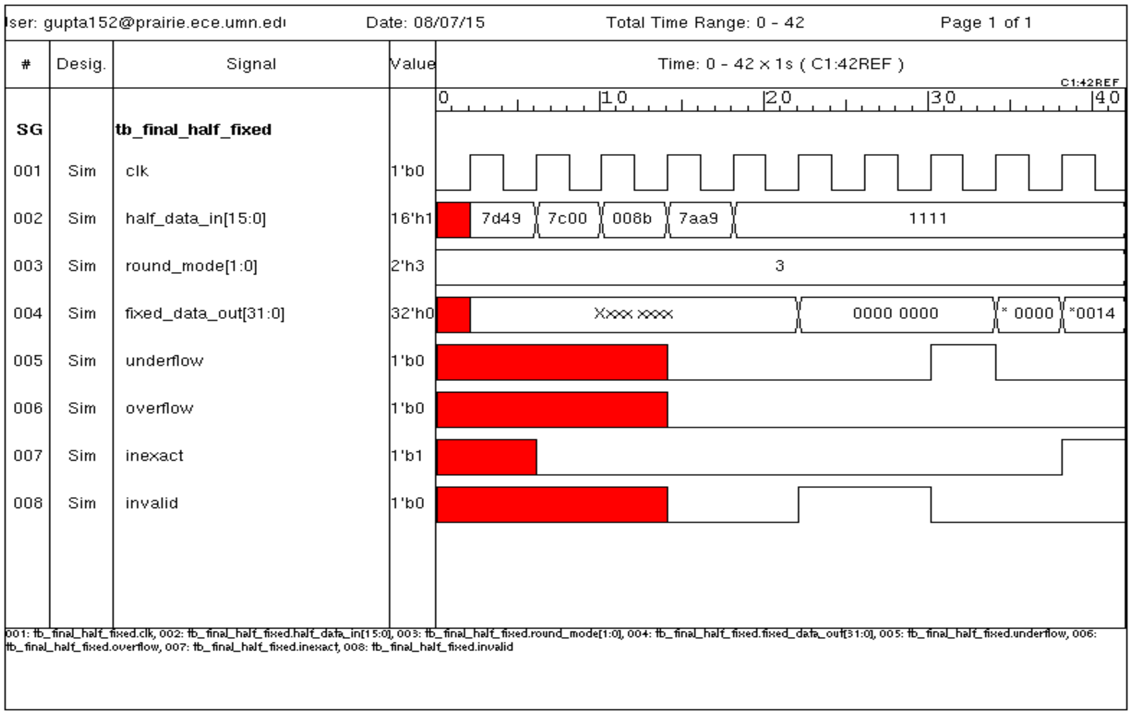

Figure 15.Waveform of half precision to fixed-point conversion

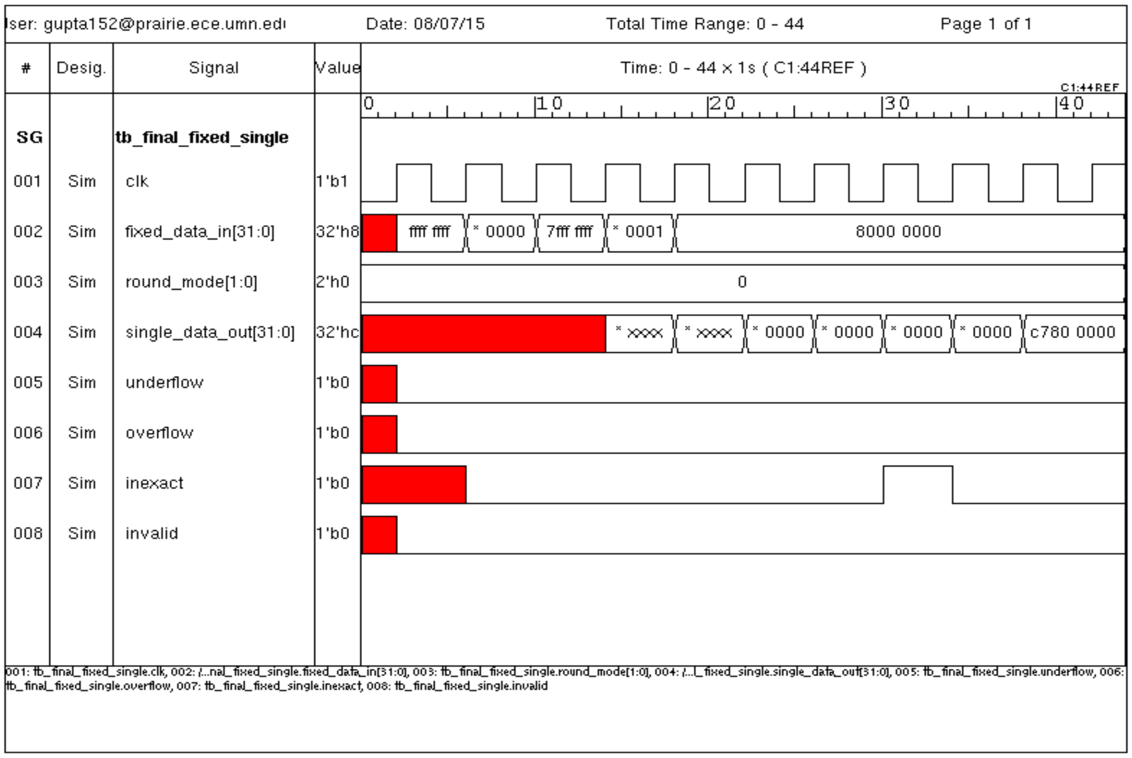
Iser: gupta152@prairie.ece.umn.edı
Date: 08/07/15
Total Time Range: 0 - 44
Page 1 of 1
#
Desig.
Signal
Value
Time: 0 - 44 x 1s (C1:44REF)
C1:44REF
0
10
20
30
40
SG
tb_final_fixed_single
001 Sim clk 1'b1
002 Sim fixed_data_in[31:0] 32'h8
ffff ffff
* 0000
7fff ffff
* 0001
8000 0000
003 Sim round_mode[1:0] 2'h0
0
004 Sim single_data_out[31:0] 32'hc
* xxxx
* xxxx
* 0000
* 0000
* 0000
* 0000
c780 0000
005 Sim underflow 1'b0
006 Sim overflow 1'b0
007 Sim inexact 1'b0
008 Sim invalid 1'b0
001: tb_final_fixed_single.clk, 002: /...nal_fixed_single.fixed_data_in[31:0], 003: tb_final_fixed_single.round_mode[1:0], 004: /...l_fixed_single.single_data_out[31:0], 005: tb_final_fixed_single.underflow, 006: tb_final_fixed_single.overflow, 007: tb_final_fixed_single.inexact, 008: tb_final_fixed_single.invalid

Figure 16.Waveform of fixed-point to single precision conversion

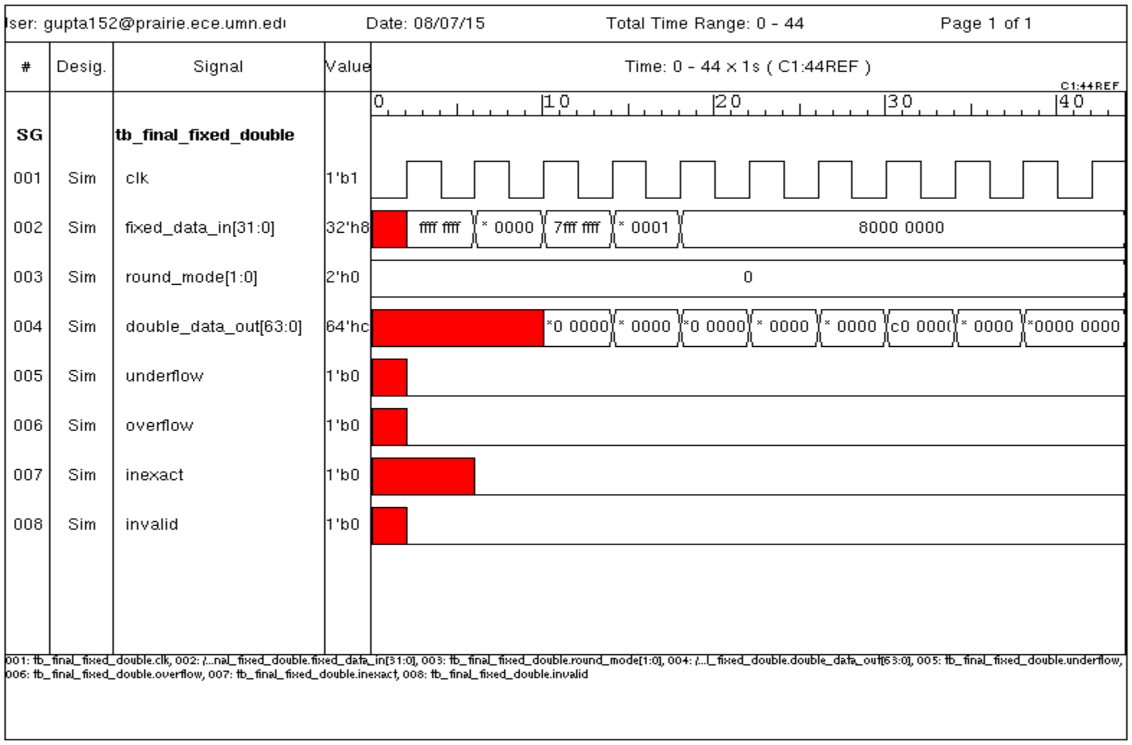
Iser: gupta152@prairie.ece.umn.edı
Date: 08/07/15
Total Time Range: 0 - 44
Page 1 of 1
#
Desig.
Signal
Value
Time: 0 - 44 x 1s (C1:44REF)
C1:44REF
0
10
20
30
40
SG
tb_final_fixed_double
001 Sim clk 1'b1
002 Sim fixed_data_in[31:0] 32'h8
ffff ffff
* 0000
7fff ffff
* 0001
8000 0000
003 Sim round_mode[1:0] 2'h0
0
004 Sim double_data_out[63:0] 64'hc
*0 0000
* 0000
*0 0000
* 0000
* 0000
c0 000(
* 0000
*0000 0000
005 Sim underflow 1'b0
006 Sim overflow 1'b0
007 Sim inexact 1'b0
008 Sim invalid 1'b0
001: tb_final_fixed_double.clk, 002: /...nal_fixed_double.fixed_data_in[31:0], 003: tb_final_fixed_double.round_mode[1:0], 004: /...l_fixed_double.double_data_out[63:0], 005: tb_final_fixed_double.underflow, 006: tb_final_fixed_double.overflow, 007: tb_final_fixed_double.inexact, 008: tb_final_fixed_double.invalid

Figure 17.Waveform of fixed-point to double precision conversion

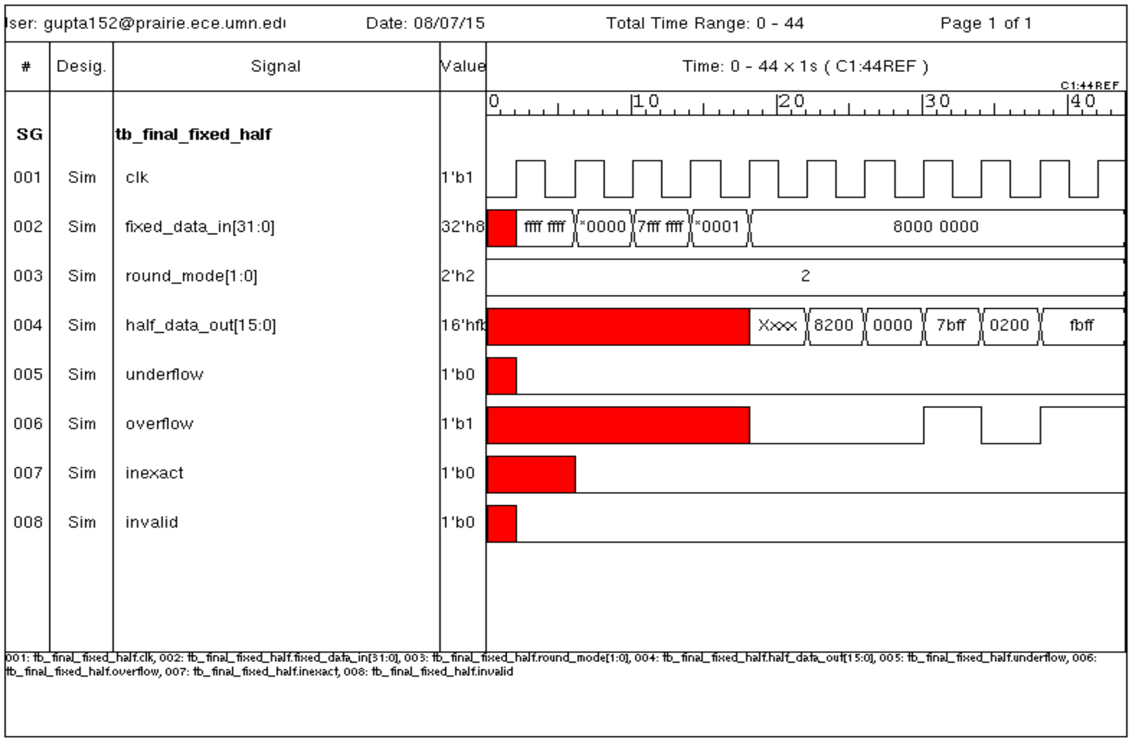
ser: gupta152@prairie.ece.umn.ed
Date: 08/07/15
Total Time Range: 0 - 44
Page 1 of 1
#
Desig.
Signal
Value
Time: 0 - 44 x 1s (C1:44REF)
C1:44REF
0
10
20
30
40
SG
tb_final_fixed_half
001 Sim clk 1'b1
002 Sim fixed_data_in[31:0] 32'h8
ffff ffff
*0000
7fff ffff
*0001
8000 0000
003 Sim round_mode[1:0] 2'h2
2
004 Sim half_data_out[15:0] 16'hfb
Xxxx
8200
0000
7bff
0200
fbff
005 Sim underflow 1'b0
006 Sim overflow 1'b1
007 Sim inexact 1'b0
008 Sim invalid 1'b0
001: tb_final_fixed_half.clk, 002: tb_final_fixed_half.fixed_data_in[31:0], 003: tb_final_fixed_half.round_mode[1:0], 004: tb_final_fixed_half.half_data_out[15:0], 005: tb_final_fixed_half.underflow, 006: tb_final_fixed_half.overflow, 007: tb_final_fixed_half.inexact, 008: tb_final_fixed_half.invalid

Figure 18.Waveform of fixed-point to half precision conversion

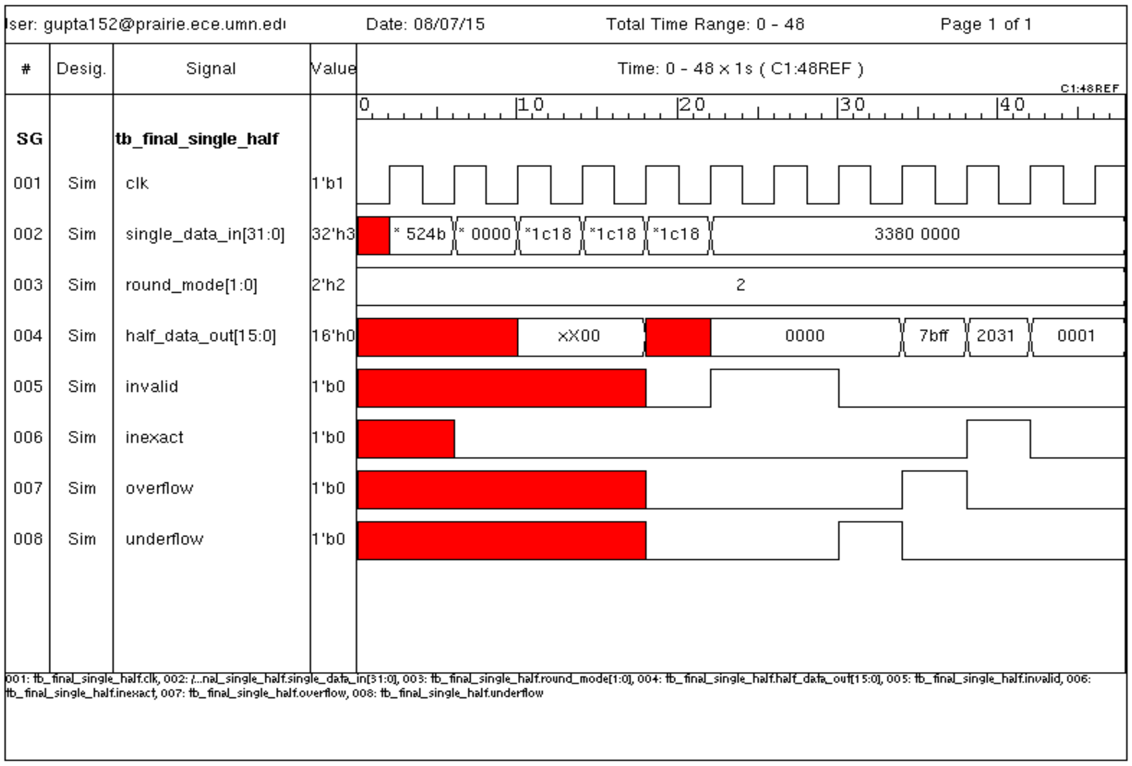
ser: gupta152@prairie.ece.umn.ed
Date: 08/07/15
Total Time Range: 0 - 48
Page 1 of 1
#
Desig.
Signal
Value
Time: 0 - 48 x 1s (C1:48REF)
C1:48REF
0
10
20
30
40
SG
tb_final_single_half
001 Sim clk 1'b1
002 Sim single_data_in[31:0] 32'h3
* 524b
* 0000
*1c18
*1c18
*1c18
3380 0000
003 Sim round_mode[1:0] 2'h2
2
004 Sim half_data_out[15:0] 16'h0
xX00
0000
7bff
2031
0001
005 Sim invalid 1'b0
006 Sim inexact 1'b0
007 Sim overflow 1'b0
008 Sim underflow 1'b0
001: tb_final_single_half.clk, 002: (...nal_single_half.single_data_in[31:0], 003: tb_final_single_half.round_mode[1:0], 004: tb_final_single_half.half_data_out[15:0], 005: tb_final_single_half.invalid, 006: tb_final_single_half.inexact, 007: tb_final_single_half.overflow, 008: tb_final_single_half.underflow

Figure 19.Waveform of single precision to half precision conversion

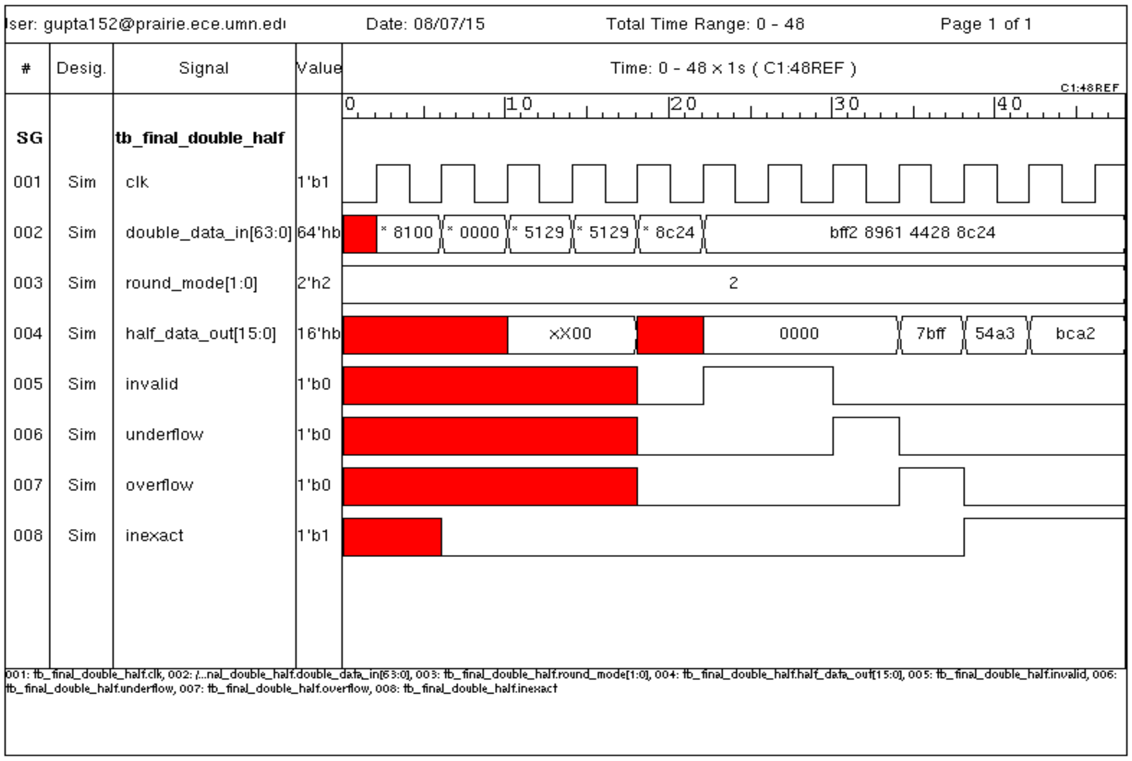

Figure 20.Waveform of double precision to half precision conversion

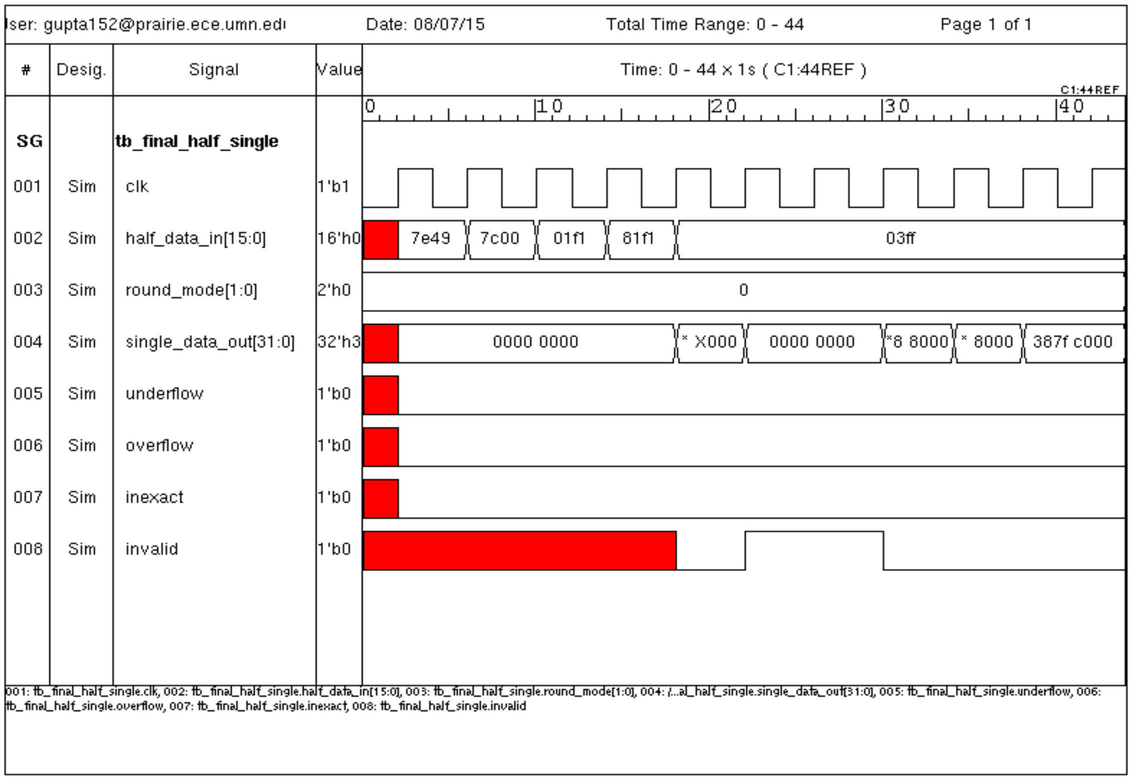

Figure 21.Waveform of half precision to single precision conversion
]

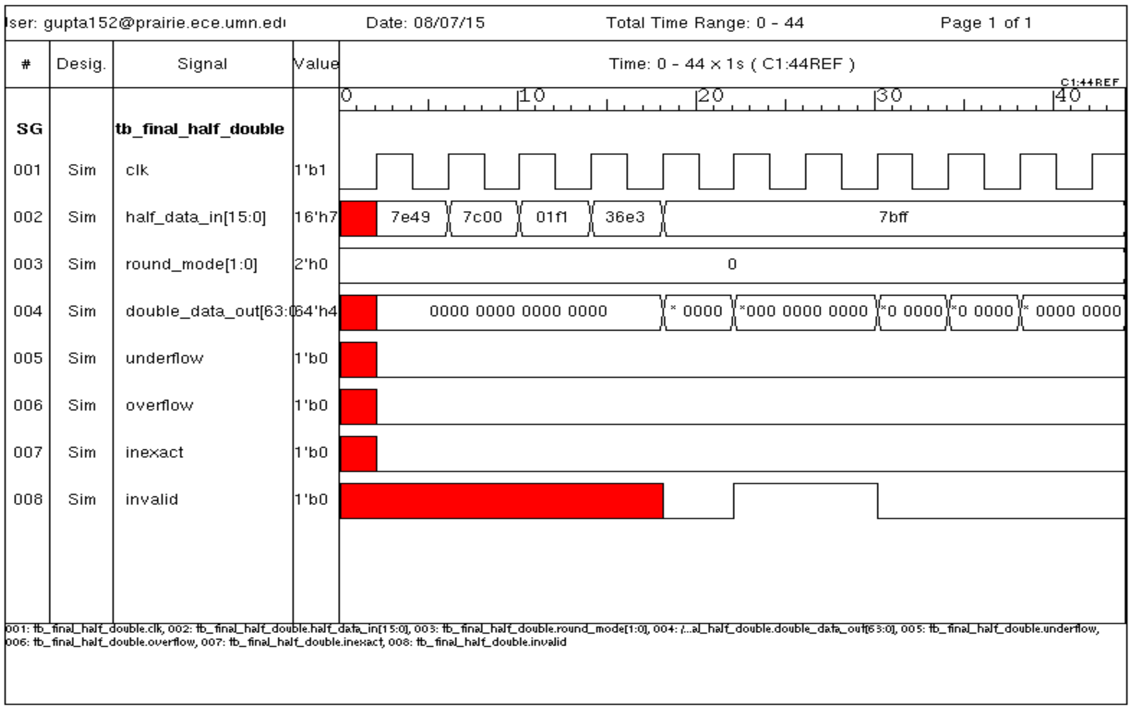
ser: gupta152@prairie.ece.umn.ed
Date: 08/07/15
Total Time Range: 0 - 44
Page 1 of 1
# Desig. Signal Value
Time: 0 - 44 x 1s (C1:44REF)
SG tb_final_half_double
001 Sim clk 1'b1
002 Sim half_data_in[15:0] 16'h7
7e49 7c00 01f1 36e3 7bff
003 Sim round_mode[1:0] 2'h0
0
004 Sim double_data_out[63:0] 64'h4
0000 0000 0000 0000 * 0000 *000 0000 0000 *0 0000 *0 0000 * 0000 0000
005 Sim underflow 1'b0
006 Sim overflow 1'b0
007 Sim inexact 1'b0
008 Sim invalid 1'b0
001: tb_final_half_double.clk, 002: tb_final_half_double.half_data_in[15:0], 003: tb_final_half_double.round_mode[1:0], 004: (...al_half_double.double_data_out[63:0], 005: tb_final_half_double.underflow, 006: tb_final_half_double.overflow, 007: tb_final_half_double.inexact, 008: tb_final_half_double.invalid

Figure 22.Waveform of half precision to double precision conversion

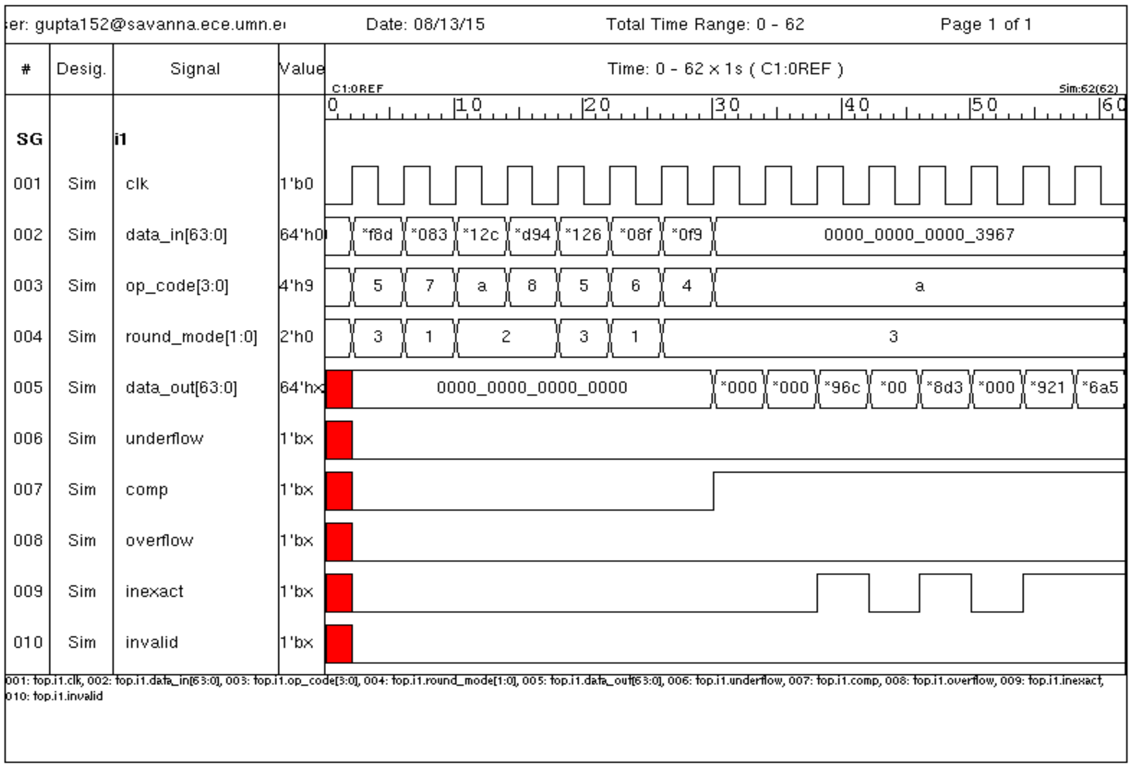
ser: gupta152@savanna.ece.umn.e
Date: 08/13/15
Total Time Range: 0 - 62
Page 1 of 1
# Desig. Signal Value
Time: 0 - 62 x 1s (C1:0REF)
SG i1
001 Sim clk 1'b0
002 Sim data_in[63:0] 64'h0
*f8d *083 *12c *d94 *126 *08f *0f9 0000_0000_0000_3967
003 Sim op_code[3:0] 4'h9
5 7 a 8 5 6 4 a
004 Sim round_mode[1:0] 2'h0
3 1 2 3 1 3
005 Sim data_out[63:0] 64'hx
0000_0000_0000_0000 *000 *000 *96c *00 *8d3 *000 *921 *6a5
006 Sim underflow 1'bx
007 Sim comp 1'bx
008 Sim overflow 1'bx
009 Sim inexact 1'bx
010 Sim invalid 1'bx
001: top.i1.clk, 002: top.i1.data_in[63:0], 003: top.i1.op_code[3:0], 004: top.i1.round_mode[1:0], 005: top.i1.data_out[63:0], 006: top.i1.underflow, 007: top.i1.comp, 008: top.i1.overflow, 009: top.i1.inexact, 010: top.i1.invalid

Figure 23.Waveform of the top-level simulation

For the top-level simulation, the input includes the data, rounding mode and the op-code, which determines the type of conversion to be performed. Also, along with the output data and exception signals another signal named *comp* is used which indicates the completion of the operation. The comp sign goes high after 7 clock cycles as expected. (5 clock cycles for the conversion modules plus the two cycles for the op-code decoder and he output de-multiplexer).

The above waveforms are for directed testbenches, and do not confirm the functionality for all the cases. For that we need to do exhaustive testing i.e. for all the possible cases of inputs. However, this is not possible practically hence we use the concepts of Verification Methodology Manual (VMM) to completely verify the design. The next section covers the verification concepts and how they are applied to test the designed IP.

6. Verification of the Design

6.1 Concepts of verification

In-order to verify or test a design a testbench needs to be developed. The functions of the testbench include,

1. Generation of the stimulus.
2. Applying that stimulus as input to the design under test (DUT).
3. Capturing the response of the design.
4. Observing the output and verifying its correctness.
5. Measuring the overall progress of verification.

As mentioned earlier, the testbench can be directed or random. In a directed testbench, the stimulus vectors are generated manually and applied as input to the DUT. The DUT is simulated and the results are manually reviewed. These results may be in the form of waveforms or log files. This type of verification using directed testbenches consumes a lot of time and resources. Also, for a complex design, it is impossible to generate all the possible input combinations and test the design completely.

Another technique of verification is done by writing constrained random testbenches. This testbench generates random stimuli and applies it as input to the DUT. Random stimulus is the most efficient way of verifying a complex design. A directed test finds bugs which are expected to be found in a design by the designer, however a random test would expose bugs which may be new and unexpected by the designer. However, on generating random stimulus, the concept of functional coverage is needed to measure the verification progress. Also, since the input vectors are randomly generated, an automated technique needs to be developed to predict the results. This is unlike the directed testbenches, since the input vectors are manually generated, the expected result is always known to the designer. Because of the above mentioned extra overhead, generating random testbenches is a very cumbersome task and building this style of verification takes much longer than the traditional directed testbench. But once, the testbench is developed, the time taken to verify the design exhaustively is much less than the time taken by directed testbenches. This can be explained by the following graph,

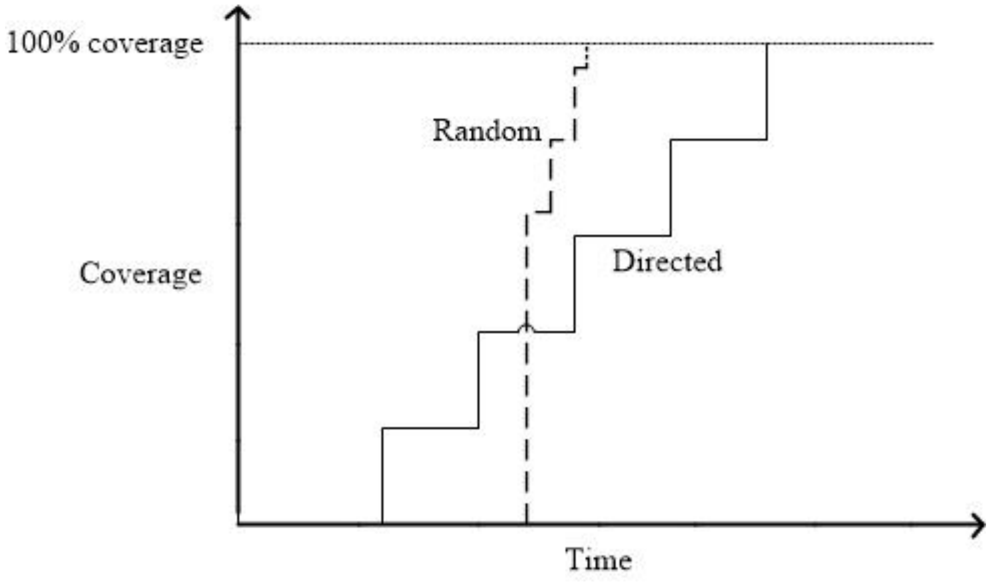

Figure 24.Directed vs. random testbench [14]

A typical exhaustive verification requires both these approaches. First a random testbench is developed and the functional coverage is measured. Different seeds are chosen to generate various random patterns of the input stimulus. Once the number of bugs found becomes constant, even though the seed is changed, a few corners can be tested using the directed testbench approach.

In a random testbench also, the input stimulus generated cannot be totally random, as the inputs generated must be valid depending on the design. Hence we need to constrain the random values generated. SystemVerilog has an inbuilt function which can help in constraining the random numbers generated. These random values are sent to the design and also to a high level model that predicts what the output should be. The outputs generated from the design are then compared with the predicted results to check for correctness.

The overall progress of the verification goals need to be measured and this is done using the functional coverage. This metric measures what features have been verified. For this special code needs to be added to the testbench to monitor the stimulus going into the DUT, its response to determine what functionality has been exercised. SystemVerilog also provides special code for this purpose.

Any complex design is verified efficiently by using a layered testbench. This means, that the testbench should not be one large piece of code. It should be divided using the tasks and functions available in SystemVerilog. This makes the verification process easy to use and maintain and also enables reusability of modules for future designs.

A block diagram describing the components of a complete testbench is as follows,

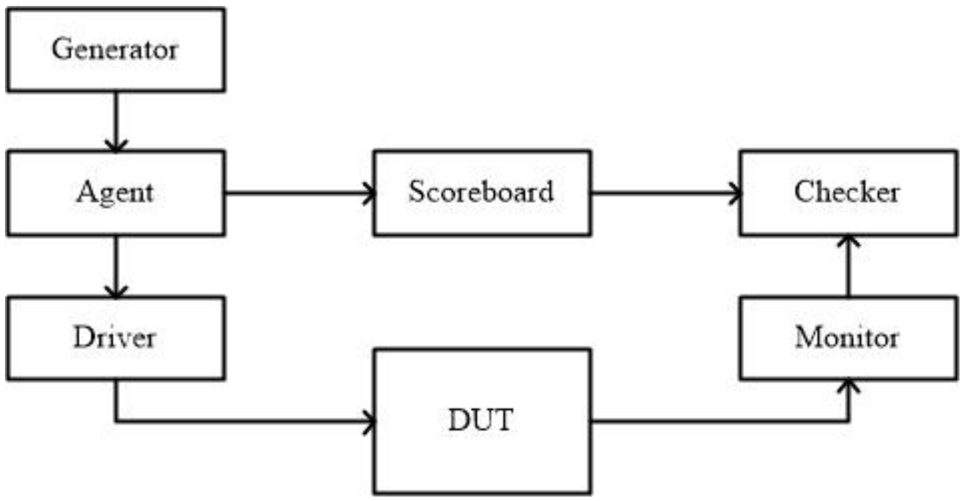

Figure 25.Components of a complete Testbench [14]

1. The design under test (DUT) accepts the inputs from the driver. The driver is responsible for simple commands which are generally at the bit level.

2. The output of the DUT drives the monitor, which accepts the signal transactions and groups them together.

3. The agent block receives higher level transactions and breaks them into individual commands or transactions. These commands are sent to the driver as well as the scoreboard. The scoreboard is responsible for predicting the expected results of the transaction.

4. The checker compares the output data from the monitor and the results predicted by the scoreboard.

5. The generator initiates the transaction and forms the highest level of abstraction.

All these transactions are then monitored by the coverage code in the testbench and the functional coverage is determined. Ideally a functional coverage of 100% indicates a completely verified design.

6.2 Communication between testbench and design

The testbench and the design need to be connected via wires in-order to exchange signals. The testbench has to connect to the design to send the input and accepts the outputs. This was done in Verilog by simple positional naming. While instantiating the design in the testbench, the order in which the signals were mentioned would imply the connections made. This is easy and simple if the design is small and has few numbers of connections, however, in a complex design there would be hundreds of connections between the testbench and the design and writing each pin while connecting the DUT and testbench would be a cumbersome task and is also error prone. Also, as a signal moves through several layers of hierarchy, it needs to be declared again and again. Adding a signal in the design would be tedious as it needs to be connected in all the modules of the design. This task of connection between DUT and testbench is highly simplified by the use of interfaces.

An interface is basically an intelligent bundle of wires, which maintains the connectivity and synchronization between two blocks or modules. An example of the interface construct is shown below. This is from the single precision to fixed-point conversion module.

```
interface single_fixed_if(input logic clk);
   logic [31:0] single_data_in, fixed_out_32;
   logic [1:0]   round_mode;
   logic   invalid, overflow, underflow, inexact;

   modport TEST (input clk,
            input fixed_out_32,
            input invalid,
            input overflow,
            input underflow,
            input inexact,
            output single_data_in,
            output round_mode
            );

   modport DUT (input clk,
           input single_data_in,
           input round_mode,
           output fixed_out_32,
           output invalid,
           output overflow,
           output underflow,
           output inexact
           );

endinterface // single_fixed_if
```

The name of the interface is single_fixed_if. It bundles the single_data_in, fixed_data_out, round_mode, invalid, overflow, underflow and inexact signals. These signals are connected between the DUT and testbench. The `modport` construct specifies the direction of each signal. The `modport TEST` specifies the direction of signals in the testbench and the `modport DUT` specifies the direction of signals in the DUT.

Now, the definition of the DUT and testbench modules will be as follows,

```
//For DUT
module final_single_fixed32 (single_fixed_if.DUT i1);
//For Testbench
program test_single_fixed32 (single_fixed_if.TEST i1);
```

The name of the interface instance is `i1`. Now, all the signals belonging to the interface `i1` will be represented using the dot (.) operator. Example,

```
i1.invalid;
i1.single_data_in;
i1.clk;
```

The top level module containing both the DUT and the testbench for the above example is written as follows,

```
module top;

   logic clk;
   initial begin
      clk = 0;
   end

   always #2 clk = ~clk;

   single_fixed_if i1(clk);
   final_single_fixed32 f1(i1);
   test_single_fixed32 t1(i1);

endmodule // top
```

6.3 Randomization

As mentioned earlier, the input stimulus to the DUT must be randomized by the testbench for efficient verification of the design. SystemVerilog provides an inbuilt technique to do so. Also, the random generated data must be constrained. An example of this is as shown,

```
class Single_Fixed_Random_Data;
   randc logic sign;
   rand logic [22:0] mantissa;
   rand logic [7:0] exponent;
   constraint s_f_corner {
      exponent < 144;
      exponent > 110;
   }
endclass
```

Here a class named `Single_Fixed_Random_Data` is defined that contains the data that needs to be randomized. In this example it is the sign, exponent and the mantissa. The rand construct tells the simulator that the variable is randomly generated. The exponent has been constrained using the construct `constraint`. According to this, the randomly generated exponent value will be less than 144, but more than 110.

In-order to generate the random values, an object of the class needs to be created and the function `randomize()` is called. This is shown below,

```
Single_Fixed_Random_Data s1;
s1 = new();
s1.randomize();
```

6.4 Functional Coverage

SystemVerilog also provides an inherent feature to measure the functional coverage. This is achieved by using the `covergroup` construct. An example of this is shown below,

```
covergroup coverage;
      coverpoint p.exponent_in;
      coverpoint p.mantissa_in;
      coverpoint p.round_mode_in;
endgroup // coverage
```

In the above example, the `covergroup` consists of three signals, exponent, mantissa and rounding mode. Depending on the width of each signal, the possible values are classified into bins. For example, the exponent for a single precision format is 8 bits wide, thus it has a total of 256 possibilities. These 256 possibilities are divided into a number of bins. The simulator then checks whether each of these bins has been covered during the simulation of the testbench. Let's assume the simulator generates 8 bins for the exponent variable. So for the exponent of single precision each bin contains 32 possibilities. During the simulation, if the generated values of exponent cover all the bins, then we achieve 100% functional coverage. This can be explained using the following example of a functional coverage report of the single to fixed conversion,

```
-----------------------------------------------------------------

Summary for Variable p.exponent_in


CATEGORY                     EXPECTED UNCOVERED COVERED PERCENT
Automatically Generated Bins 8        1         7       87.50


Automatically Generated Bins for p.exponent_in


Uncovered bins

NAME           COUNT AT LEAST NUMBER
[auto[96:127]] 0     1        1


Covered bins

NAME          COUNT AT LEAST
auto[0:31]    1     1
auto[32:63]   1     1
auto[64:95]   1     1
auto[128:159] 2     1
auto[160:191] 2     1
auto[192:223] 2     1
auto[224:255] 1     1

-----------------------------------------------------------------
```

In the above report, the number of bins created for the exponent variable is 8. Hence the 256 possible combinations are divided into 8 bins where each bin contains 32 possibilities. One of the bin in uncovered in the above simulation, as the value of the randomly generated exponent was never between 96 and 127. Hence the coverage for the exponent variable is 87.50 %.

As the number of bins is increased, for a given number random data generated, the functional coverage percentage decreases. This is because more the number of bins, the number of possibilities in each bin decreases. The following graph shows the functional coverage of the exponent variable as a function of the number of bins for 100 trials.

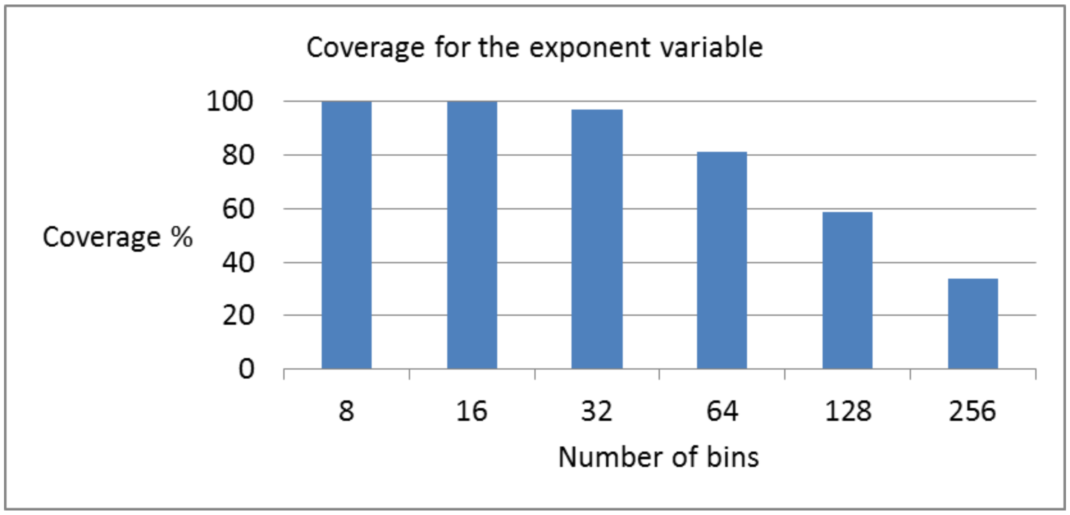

Figure 26.Graph showing the coverage as a function of number of bins

The variables in the `covergroup` can be sampled using the `sample()` function.

```
coverage c1;
c1 = new();
c1.sample();
```

The following sections include the algorithms used to verify the individual modules of the IP design. A bottom-up approach was used to verify the design, i.e. the individual modules were verified first and then the top level was verified.

6.5 Floating to fixed-point verification algorithm

1. Generate a set of random data including the sign, exponent, fraction and the rounding mode. The width of the exponent and fraction depends on the floating point format (single, double or half). The block performing this function acts as the agent of the

testbench, discussed earlier. This set of randomly generated data is passed onto the driver as well as the scoreboard.

2. The driver block, sends the data from the agent to the DUT.

3. In the scoreboard, based on the exponent value, the number is classified as NaN, invalid or zero. Also, it is predicted, whether an overflow or underflow exception will be generated. Accordingly, the expected result from the DUT is predicted and stored in an array.

4. If the number is normal and no exception signals are generated then the prediction of the expected result is done by the scoreboard. First, the value of the floating point number is calculated by using the sign, exponent and fraction from the agent. For double precision format an inbuilt function is available `$bitstoreal`, which accepts the concatenated sign, exponent and fraction bits (64-bit) and converts it into a real value. However, for single precision format (32 bit) a separate function was developed in C and was accessed using DPI (Direct Programming Interface). This function accepts the sign, exponent and fraction of the single precision representation and returns the real value.

5. Once the value is available, it is rounded off to the closest integer which is smaller in magnitude. Predict all the values possible in the fixed-point representation (Q17.15 format) from that rounded off number to the next larger consecutive integer and store it in an array. Example if the real value is 8.976, then it is rounded off to 8. All the Q17.15 format representations between 8 and 9 are stored in an array. This is possible because the precision in fixed-point numbers is constant (0.000030517578125 in this case).

6. Find out the two values from this array between which the input real number lies. These two numbers represent the possible outputs of the DUT. From these two numbers the expected result is predicted based on the rounding mode. All these predicted results are stored in an array.

7. Wait for five clock cycles and then sample the outputs of the DUT. The output of the DUT will be 32 bits (Q17.15 format). This needs to be converted into a real value, which is done by using a function. The real values obtained from the DUT are also stored in an array.

8. The checker compares the two arrays, one containing the predicted results and the other containing the DUT results. This confirms the functionality of the module.

The above algorithm is explained using the following flow graph,

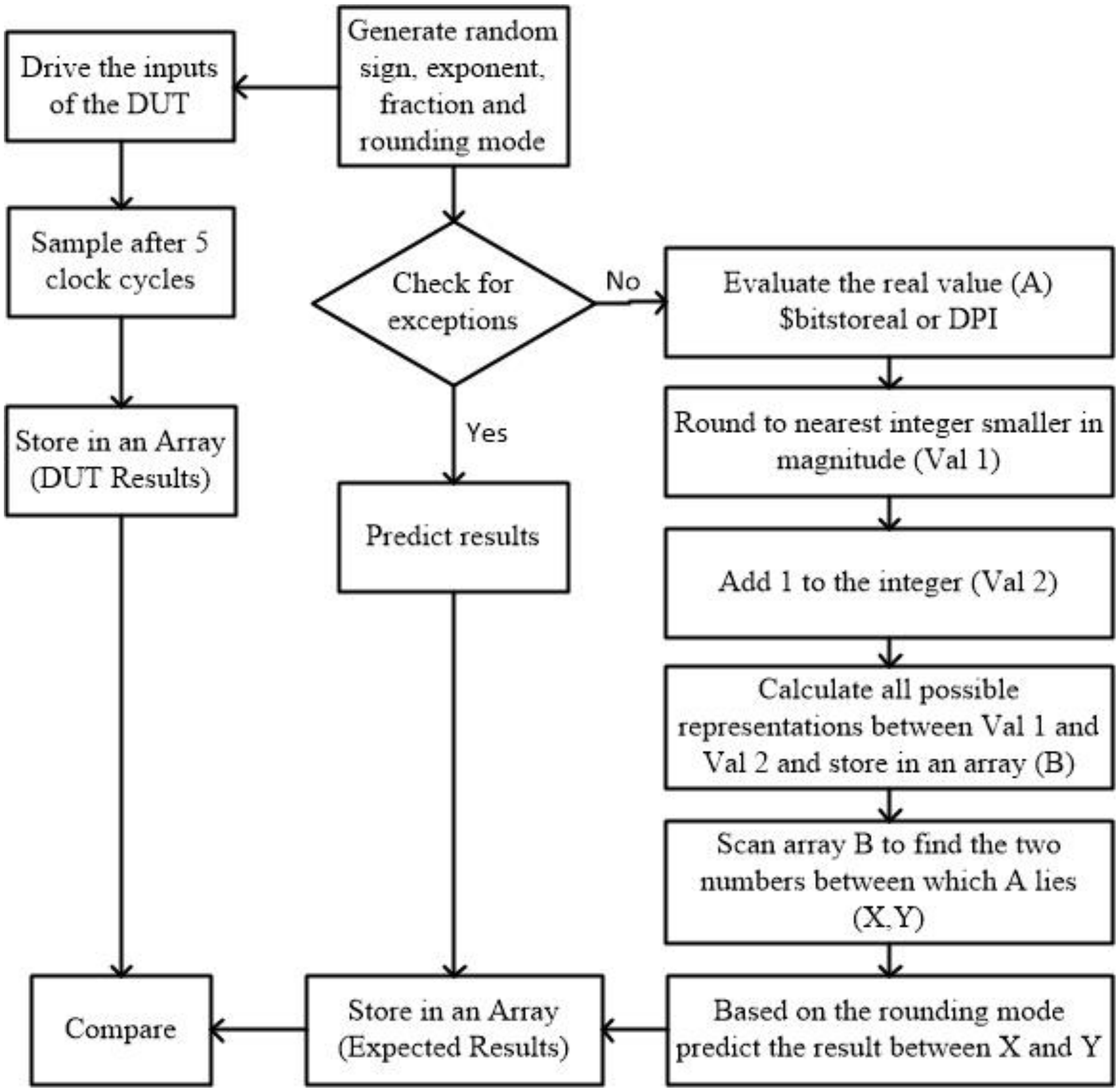
Generate random sign, exponent, fraction and rounding mode
Drive the inputs of the DUT
Sample after 5 clock cycles
Store in an Array (DUT Results)
Check for exceptions
No
Yes
Predict results
Evaluate the real value (A) $bitstoreal or DPI
Round to nearest integer smaller in magnitude (Val 1)
Add 1 to the integer (Val 2)
Calculate all possible representations between Val 1 and Val 2 and store in an array (B)
Scan array B to find the two numbers between which A lies (X,Y)
Based on the rounding mode predict the result between X and Y
Store in an Array (Expected Results)
Compare

Figure 27.Floating to fixed-point verification algorithm

6.6 Fixed-point to floating point verification algorithm

1. Generate the random input data i.e. the Q17.15 fixed point format (32 bits), along with the rounding mode. This acts as the agent of the testbench. The input is driven to the DUT.

2. Calculate the value of the generated fixed-point format. This is done using a function written in SystemVerilog. This is used by the scoreboard to predict the output of the DUT.

3. Convert the fixed value to the 64 bit double precision floating point format using the system task `$realtobits`. This gives the predicted value for fixed-point to double precision floating point conversion. For the fixed-point to single and half precision conversion, the obtained double precision number is converted to single precision or half precision depending on the operation. This can be performed by truncating the fraction and adjusting the exponent bias. This result gives one of the possible outputs of the DUT, which is either equal to or less than the actual input value.

4. However another output is possible which is obtained by incrementing the fraction by one. The input real value lies between these two possible outputs and depending on the rounding mode one the two possibilities is chosen to be the predicted output. The predicted outputs are stored in an array.

5. The monitor block samples the output of the DUT after 5 clock cycle and stores the result in an array. The checker block compares the two arrays containing the DUT results and the predicted outputs.

The above algorithm can be explained using the following flow chart,

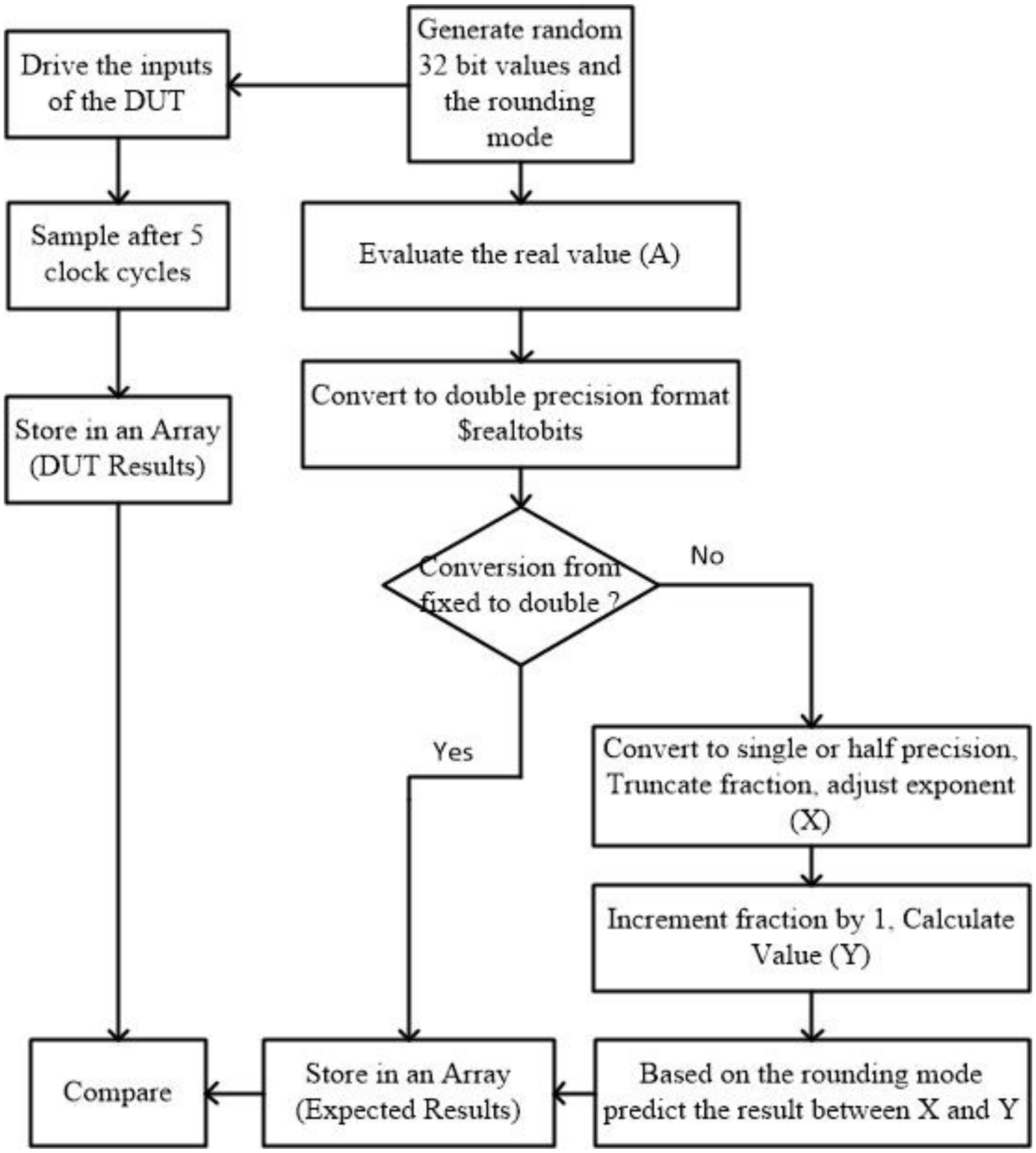
Generate random 32 bit values and the rounding mode
Drive the inputs of the DUT
Sample after 5 clock cycles
Store in an Array (DUT Results)
Evaluate the real value (A)
Convert to double precision format $realtobits
Conversion from fixed to double ?
No
Yes
Convert to single or half precision, Truncate fraction, adjust exponent (X)
Increment fraction by 1, Calculate Value (Y)
Based on the rounding mode predict the result between X and Y
Store in an Array (Expected Results)
Compare

Figure 28.Fixed-point to floating point verification algorithm

6.7 Single and Double precision to half precision verification algorithm

1. Generate the random input data, including the sign, exponent, fraction and rounding modes. This is generated by the agent and driven to the DUT by the driver.

2. The scoreboard uses the exponent values to classify the number as invalid, NaN or denormal number. It also evaluates whether there is an overflow or underflow exception. Depending on the exception raised, it predicts the expected output of the DUT and stores the result in an array.

3. If no exceptions are raised, then the real value is calculated by using `$bitstoreal` for double precision or the DPI function for single precision. The real value is then converted to double precision format using `$realtobits`. This value is then converted to half precision format by adjusting the fraction and exponent. This value is one of the possible outputs of the DUT.

4. The other possible output is calculated by incrementing the fraction by 1. The input real value lies between these two possible outputs predicted by the scoreboard. Depending on the rounding mode one of these two outputs is predicted as the expected result of the DUT. These predicted results are stored in an array.

5. After 5 clock cycles, the monitor samples the output of the DUT and stores the result in another array. Both the arrays are compared by the checker to confirm the functionality.

The above algorithm can be explained using the following flow graph,

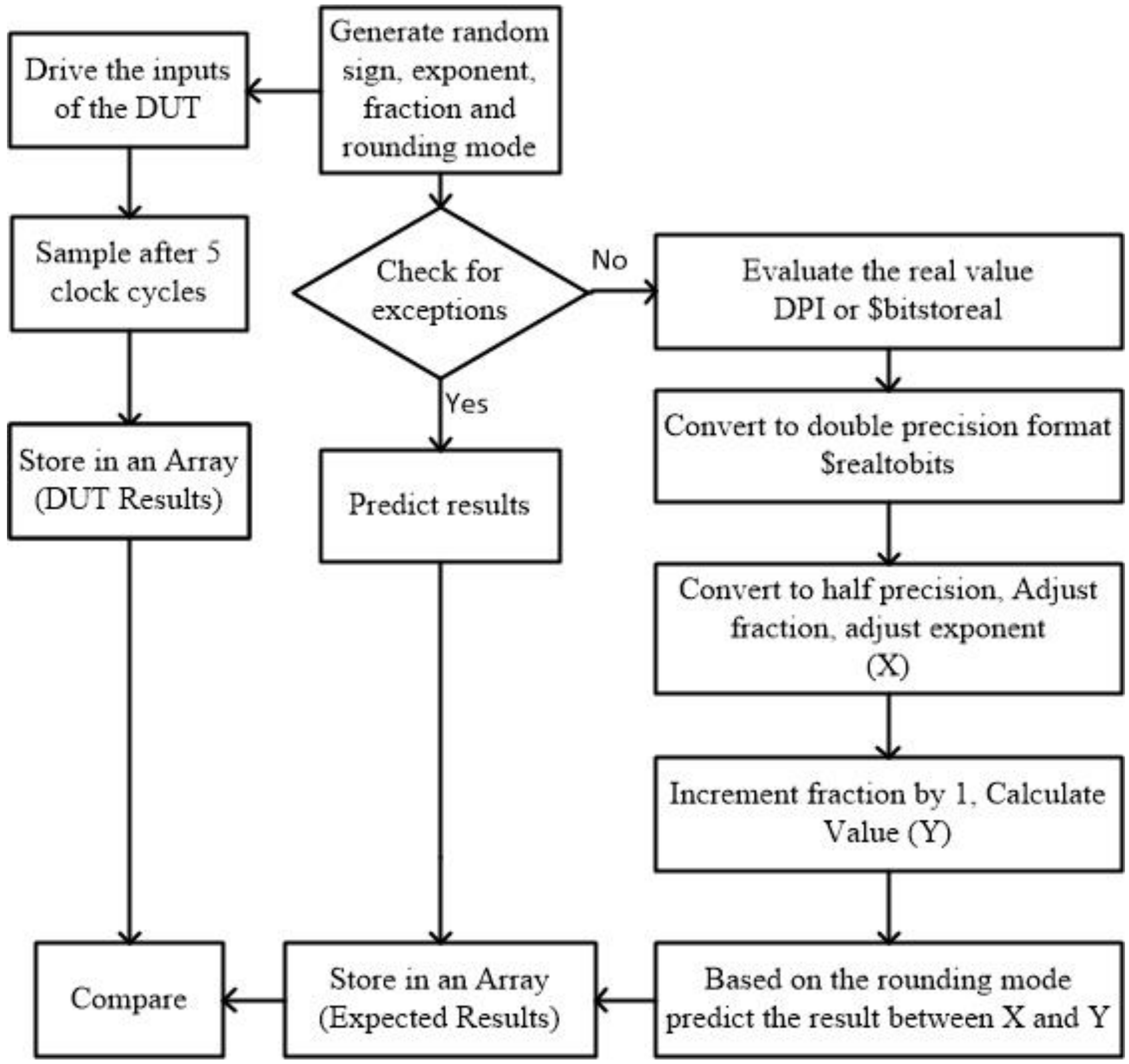
Generate random sign, exponent, fraction and rounding mode
Drive the inputs of the DUT
Sample after 5 clock cycles
Store in an Array (DUT Results)
Check for exceptions
No
Yes
Evaluate the real value DPI or $bitstoreal
Convert to double precision format $realtobits
Convert to half precision, Adjust fraction, adjust exponent (X)
Increment fraction by 1, Calculate Value (Y)
Predict results
Based on the rounding mode predict the result between X and Y
Store in an Array (Expected Results)
Compare

Figure 29.Single/Double precision to half precision verification algorithm

6.8 Half precision to Single and Double precision verification algorithm

1. Generate the random input vectors including the sign, exponent, fraction and rounding modes. This acts as the agent block. The driver sends these signals to the DUT as inputs.

2. The scoreboard uses the exponent to classify the numbers as NaN, infinity or zero. According to the exceptions generated the output of the DUT is predicted and stored in an array.

3. If none of the exceptions is generated, the scoreboard needs to predict the output of the DUT. To do this, the input sign, exponent and fraction are concatenated and converted to a real value using a C function imported by DPI.

4. Since all the numbers represented by half precision can be expressed using single or double precision numbers, the input real number is the predicted output of the DUT and is stored in an array.

5. The monitor samples the output of the DUT after 5 clock cycles and stores the result in an array. The checker compares the two arrays to check the module for functionality.

The algorithm can be expressed using the following flow chart,

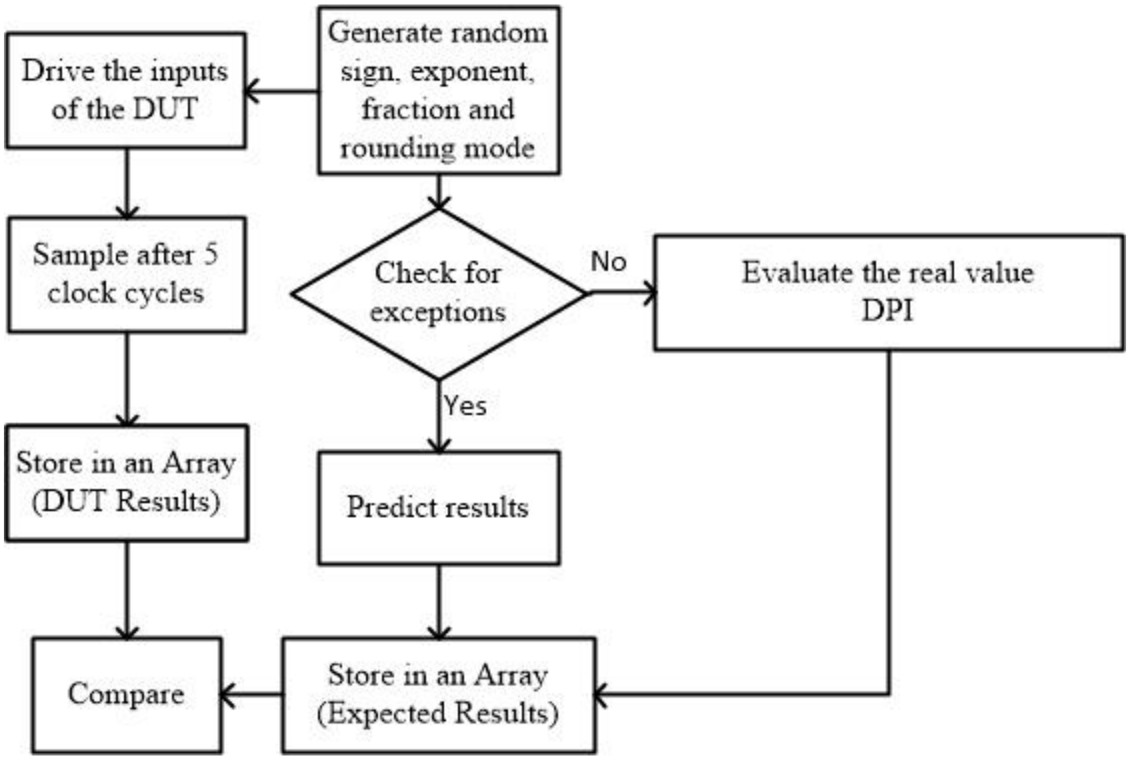
Generate random sign, exponent, fraction and rounding mode
Drive the inputs of the DUT
Check for exceptions
No
Evaluate the real value DPI
Sample after 5 clock cycles
Yes
Store in an Array (DUT Results)
Predict results
Compare
Store in an Array (Expected Results)

Figure 30.Half precision to Single/Double precision verification algorithm.

6.9 Verification of the top-level architecture

The top-level verification testbench is developed by incorporating all the testbench components of the individual modules. (Drivers, Monitors, Checkers, Scoreboards, Agents) The algorithm is explained as follows,

1. Generate the random data acting as input to the top-level IP. The input data includes the op-code, rounding mode and the reset signal. Depending on the op-code, the control is transferred to the agent of the corresponding conversion module, which generates the corresponding random input data. The driver at the top level accepts the input from the agent and packs it into the required 64-bit format and feeds it to the IP.

2. The top-level scoreboard depending on the op-code, transfers control to the scoreboard of the corresponding conversion module, and the output result of the DUT is predicted. These predictions are returned to the top-level scoreboard, where they are stored in an array.

3. The top-level monitor samples the 64-bit output of the IP and converts it to the required format depending on the op-code. The corresponding conversion module, then converts the bit representation to a real value which is stored in another array.

4. The checker block compares the two arrays to check for functionality.

5. The functional coverage is measured by considering the op-code and the rounding mode.

7. Verification Results

The verification results of the individual modules can be summarized in the following table. The table lists the variables for which the functional coverage was measured depending on the conversion module and the number of bins into which each variable is divided. The number of cases refers to the number of input combinations tried in order to achieve 100% functional coverage.

Table 12.Verification results of the individual modules

Module	**Variables to measure functional coverage**	**Number of bins**	**Number of cases**
Single to Fixed	Sign (1 bit)	2	1500
	Exponent (8 bit)	256	
	Fraction (23 bits)	128	
	Rounding mode (2 bits)	4	
Double to Fixed	Sign (1 bit)	2	3000
	Exponent (10 bits)	512	
	Fraction (53 bits)	512	
	Rounding mode (2 bits)	4	
Half to Fixed	Sign (1 bit)	2	500
	Exponent (5 bits)	32	
	Fraction (10 bits)	64	
	Rounding mode (2 bits)	4	
Fixed to half	Q17.15 format (32 bits)	64	250
	Rounding mode (2 bits)	4	
Fixed to Single	Q17.15 format (32 bits)	128	600
	Rounding mode (2 bits)	4	
Fixed to Double	Q17.15 format (32 bits)	256	1500
	Rounding mode (2 bits)	4	

Single to half	Sign (1 bit)	2	1500
	Exponent (8 bits)	256	
	Fraction (23 bits)	128	
	Rounding mode (2 bits)	4	
Double to half	Sign (1 bit)	2	3000
	Exponent (10 bits)	512	
	Fraction (53 bits)	512	
	Rounding mode (2 bits)	4	
Half to Double	Sign (1 bit)	2	500
	Exponent (5 bits)	32	
	Fraction (10 bits)	64	
	Rounding mode (2 bits)	4	
Half to single	Sign (1 bit)	2	800
	Exponent (5 bits)	32	
	Fraction (10 bits)	128	
	Rounding mode (2 bits)	4	

The verification results of the top-level module is as shown below,

Table 13.Verification results of the top-level module

Module	**Variables to measure functional coverage**	**Number of bins**	**Number of trials**
Top-level	Op-code (4 bits)	16	500
	Rounding mode (2 bits)	4	
	Reset	2	

8. Comparison with the software implementation

The designed IP represents a hardware level implementation of the conversions between the floating and fixed point formats. This section focuses on the comparison of the number of cycles taken by the hardware implementation versus the number of cycles taken by a compiler to complete the conversion in software.

To evaluate the number of cycles for a software implementation, the MPLAB C30 compiler for the PIC microcontroller was chosen. The `dsp.h` header file in the compiler consists of two inbuilt functions written in C to perform the conversions between the floating and the fixed point formats. These files are named as `flt2frct.c` and `frct2flt.c` which perform the floating to fixed point and fixed to floating point conversion respectively.

These C files were compiled and the generated assembly listing was observed. The number of assembly instructions would approximately give us the number of cycles needed to perform the conversion operation. The assumption made in this case is that one instruction takes one cycle to complete. The following table lists the number of cycles for each conversion operation.

Table 14. Number of cycles for each conversion operation in software

Conversion	Number of cycles (approx.)
Floating to fixed point	226
Fixed to floating point	38

From the simulation results of the proposed hardware implementation which is a 7-stage pipeline, the number of cycles for each conversion in that case will be 7, which implies that the hardware implementation causes tremendous speed-up for the conversions as compared to the software implementation.

The C file including the conversion functions and their corresponding assembly listing is attached in the appendix at the end of the report.

9. Conclusion

In this research thesis, a floating point conversion IP was proposed. It also included the newly introduced half-precision floating point format, which holds tremendous promise for future applications involving floating point and also might replace the traditional single and double precision floating point formats which are used currently. The IP is fully compliant with the IEEE 754-2008 standard including the rounding modes and the exception signals. The design was implemented using SystemVerilog and its verification was completed using the concepts of VMM which is an industry standard for verification.

Future work may include synthesizing and optimizing the design of the IP. Many modules have common stages which may be combined into one to reduce the area and power consumption. The IP conversion units may also be modified to perform additional functions like simple arithmetic operations. The conversion modules currently use the basic algorithms for conversions between the various floating point and fixed point formats. More advanced or optimized algorithms may be implemented to improve throughput and efficiency of the conversion IP.

9. References

[1] IEEE standards board and ANSI. IEEE Standards for Binary Floating-point Arithmetic, 2008, IEEE Std 754-2008.

[2] Aneesh R, Vinayak Patil, Sobha PM, A David selvakumar, “HMFPCC:- Hybrid-Mode Floating Point Conversion IP”, 2015 International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA).

[3] D. Menard, D.Chillet, F.Charot, O.Sentieys, “Automatic Floating point to Fixed-point Conversion for DSP Code generation”, in proc. of the International Conference on Compilers, Architectures, and synthesis for Embedded systems, Oct. 2002.

[4] C. Shi, R.W. Brodersen, “An Automated floating point to Fixed point Conversion methodology ”, in proc. of IEEE International Conferences on Acoustics, speech and signal processing.

[5] Pavle Belanovic, Markus Rupp “Automated Floating-point to Fixed-point conversion with the fixify environment”, in proc. of 16th international workshop on Rapid System Prototyping.

[6] D Williamson, “Dynamically scaled fixed-point arithmetic” proc. IEEE Pacific Rim Conf. on communication, computation and signal processing, Vol. 1, 1991.

[7] S. Kim and W. Sung, “A Floating-point to Fixed-point Assembly Program Translator for the TMS320C25”, IEEE Transactions on Circuits and Systems, Vol. 41, Nov. 1994.

[8] Clemens Maaß, Matthias Baer and Marc Kachelrieß, “CT image reconstruction with half-precision floating point values” IEEE Medical Imaging Conference Record, 2009.

[9] Alexandru Barleanu, Vadim Baitoiu, Anderi Stan, “” Floating-point to fixed-point code conversion with variable trade-off between computational complexity and accuracy loss”, in proc. of 15th international conference on system theory, control and computing, 2011.

[10] The Scientist and Engineer’s guide to Digital Signal Processing by Steven W. Smith, PhD.

[11] Fixed-Point vs. Floating-point Digital Signal Processing – Analog Devices.

[12] Comparing Fixed- and Floating-Point DSPs by Gene Frantz and Ray Simar, Texas instruments.

[13] Right-sizing Precision for speed and power efficiency by Dr. John L. Gustafson, Director, Intel Labs, Intel Developer Forum, 2012.

[14] SystemVerilog for Verification – A Guide to Learning the Testbench Language Features by Chris Spear and Greg Tumbush.

Appendix

1. C code and assembly listing for frct2flt.c

```
/* Local headers. */
#include "dsp.h"
float Fract2Float ( /* Converts fractional into float */
       fractional aVal /* fract value in range {-1, 1-2^-15} */
) {
 2EC  FA0012     lnk #0x12
 2EE  980F00     mov.w 0x0000,[0x001c+16]

              #if   DATA_TYPE==FRACTIONAL       /* [ */

                 /* Local declarations. */
                 double scale = pow (2.0, -15.0); /* 2^(-15) */
 2F0  200000     mov.w #0x0,0x0000
 2F2  238001     mov.w #0x3800,0x0002
 2F4  980760     mov.w 0x0000,[0x001c+12]
 2F6  980771     mov.w 0x0002,[0x001c+14]
                 long int fullRange = 1L<<16;  /* 2^(16) */
 2F8  200000     mov.w #0x0,0x0000
 2FA  200011     mov.w #0x1,0x0002
 2FC  980740     mov.w 0x0000,[0x001c+8]
 2FE  980751     mov.w 0x0002,[0x001c+10]
                 long int halfRange = 1L<<15;  /* 2^(15) */
 300  280000     mov.w #0x8000,0x0000
 302  200001     mov.w #0x0,0x0002
 304  980720     mov.w 0x0000,[0x001c+4]
 306  980731     mov.w 0x0002,[0x001c+6]
                 double decimalVal = 0.0;
 308  B80060     mul.uu 0x0000,#0,0x0000
 30A  BE8F00     mov.d 0x0000,[0x001c]
```

```
                /* Convert. */
                if (aVal >= halfRange) {
30C  90080E     mov.w [0x001c+16],0x0000
30E  B90161     mul.su 0x0000,#1,0x0004
310  90002E     mov.w [0x001c+4],0x0000
312  9000BE     mov.w [0x001c+6],0x0002
314  510F80     sub.w 0x0004,0x0000,[0x001e]
316  598F81     subb.w 0x0006,0x0002,[0x001e]
318  350004     bra lts, 0x000322
                   aVal -= fullRange;
31A  90004E     mov.w [0x001c+8],0x0000
31C  90088E     mov.w [0x001c+16],0x0002
31E  508000     sub.w 0x0002,0x0000,0x0000
320  980F00     mov.w 0x0000,[0x001c+16]
                }
                decimalVal = ((double) aVal)*scale;
322  90080E     mov.w [0x001c+16],0x0000
324  DE80CF     asr 0x0000,#15,0x0002
326  07FF30     rcall 0x000188
328  90016E     mov.w [0x001c+12],0x0004
32A  9001FE     mov.w [0x001c+14],0x0006
32C  07FF58     rcall 0x0001de
32E  BE8F00     mov.d 0x0000,[0x001c]

                /* Return decimal value in floating point. */
                return ((float) decimalVal);
330  BE001E     mov.d [0x001c],0x0000

             #else /* ] [ */

                /* Return input value in floating point. */
                return ((float) aVal);
             #endif     /* ] */

             } /* end of Fract2Float */
332  FA8000     ulnk
334  060000     return
```

2. C code and assembly listing for flt2frct.c

```
/* Local headers. */
#include "dsp.h"                   /* DSP Library interface */

/* Local defines. */
          #define  SCALE      1L<<15                /* 2^15 */
          #define  RANGE      1L<<16                /* 2^16 */

             /* Float2Fract implementation. */

             #if   DATA_TYPE==FRACTIONAL      /* [ */

 fractional Float2Fract (/* Converts float into fractional */
                float aVal /* float value in range [-1, 1) */
             ) {
51E  FA0018     lnk #0x18
520  BE9F88     mov.d 0x0010,[0x001e++]
522  980F10     mov.w 0x0000,[0x001c+18]
524  980F21     mov.w 0x0002,[0x001c+20]

                /* Local declarations. */
                long int scale = SCALE;
526  280000     mov.w #0x8000,0x0000
528  200001     mov.w #0x0,0x0002
52A  980770     mov.w 0x0000,[0x001c+14]
```

```
52C  980F01      mov.w 0x0002,[0x001c+16]
                 long int fractVal = 0.0;
52E  B80060      mul.uu 0x0000,#0,0x0000
530  980750      mov.w 0x0000,[0x001c+10]
532  980761      mov.w 0x0002,[0x001c+12]
                 double decimalVal = 0.0;
534  B80060      mul.uu 0x0000,#0,0x0000
536  980730      mov.w 0x0000,[0x001c+6]
538  980741      mov.w 0x0002,[0x001c+8]
                 double dummy = 0.0;
53A  B80060      mul.uu 0x0000,#0,0x0000
53C  980710      mov.w 0x0000,[0x001c+2]
53E  980721      mov.w 0x0002,[0x001c+4]
                 int isOdd = 0;
540  EB0000      clr.w 0x0000
542  780F00      mov.w 0x0000,[0x001c]

         /* Convert with convergent rounding and saturation. */
                 decimalVal = aVal*scale;
544  90007E      mov.w [0x001c+14],0x0000
546  90088E      mov.w [0x001c+16],0x0002
548  07FE1F      rcall 0x000188
54A  90091E      mov.w [0x001c+18],0x0004
54C  9009AE      mov.w [0x001c+20],0x0006
54E  07FE47      rcall 0x0001de
550  980730      mov.w 0x0000,[0x001c+6]
552  980741      mov.w 0x0002,[0x001c+8]
                 if (aVal >= 0) {
554  B81160      mul.uu 0x0004,#0,0x0004
556  90081E      mov.w [0x001c+18],0x0000
558  9008AE      mov.w [0x001c+20],0x0002
55A  07FE75      rcall 0x000246
55C  E00000      cp0.w 0x0000
```

```
   55E  3D0001     bra ges, 0x000562
   560  370056     bra 0x00060e
                      fractVal = floor (decimalVal);
   562  90003E     mov.w [0x001c+6],0x0000
   564  9000CE     mov.w [0x001c+8],0x0002
   566  07FE71     rcall 0x00024a
   568  07FE78     rcall 0x00025a
   56A  980750     mov.w 0x0000,[0x001c+10]
   56C  980761     mov.w 0x0002,[0x001c+12]
                      dummy = fractVal/2.0;
   56E  90005E     mov.w [0x001c+10],0x0000
   570  9000EE     mov.w [0x001c+12],0x0002
   572  07FE0A     rcall 0x000188
   574  200002     mov.w #0x0,0x0004
   576  240003     mov.w #0x4000,0x0006
   578  07FE91     rcall 0x00029c
   57A  980710     mov.w 0x0000,[0x001c+2]
   57C  980721     mov.w 0x0002,[0x001c+4]
                      isOdd = (int) ((dummy - floor
(dummy))*2.0);
   57E  90041E     mov.w [0x001c+2],0x0010
   580  9004AE     mov.w [0x001c+4],0x0012
   582  90001E     mov.w [0x001c+2],0x0000
   584  9000AE     mov.w [0x001c+4],0x0002
   586  07FE61     rcall 0x00024a
   588  BE0100     mov.d 0x0000,0x0004
   58A  BE0008     mov.d 0x0010,0x0000
   58C  07FEC9     rcall 0x000320
   58E  BE0100     mov.d 0x0000,0x0004
   590  07FEC8     rcall 0x000322
   592  07FE63     rcall 0x00025a
   594  780F00     mov.w 0x0000,[0x001c]
                      dummy = decimalVal -fractVal;
   596  90005E     mov.w [0x001c+10],0x0000
   598  9000EE     mov.w [0x001c+12],0x0002
   59A  07FDF6     rcall 0x000188
   59C  BE0100     mov.d 0x0000,0x0004
   59E  90003E     mov.w [0x001c+6],0x0000
   5A0  9000CE     mov.w [0x001c+8],0x0002
   5A2  07FEBE     rcall 0x000320
   5A4  980710     mov.w 0x0000,[0x001c+2]
   5A6  980721     mov.w 0x0002,[0x001c+4]
                      if ((dummy > 0.5) || ((dummy == 0.5) &&
isOdd)) {
   5A8  200002     mov.w #0x0,0x0004
   5AA  23F003     mov.w #0x3f00,0x0006
   5AC  90001E     mov.w [0x001c+2],0x0000
   5AE  9000AE     mov.w [0x001c+4],0x0002
   5B0  07FE4A     rcall 0x000246
   5B2  E00000     cp0.w 0x0000
   5B4  3C0012     bra gts, 0x0005da
   5B6  EB4000     clr.b 0x0000
   5B8  985770     mov.b 0x0000,[0x001c+23]
   5BA  200002     mov.w #0x0,0x0004
   5BC  23F003     mov.w #0x3f00,0x0006
   5BE  90001E     mov.w [0x001c+2],0x0000
   5C0  9000AE     mov.w [0x001c+4],0x0002
   5C2  07FF0C     rcall 0x0003dc
   5C4  E00000     cp0.w 0x0000
   5C6  320001     bra z, 0x0005ca
   5C8  370002     bra 0x0005ce
   5CA  B3C010     mov.b #0x1,0x0000
   5CC  985770     mov.b 0x0000,[0x001c+23]
   5CE  90507E     mov.b [0x001c+23],0x0000
   5D0  A20400     btg 0x0000,#0
   5D2  E00400     cp0.b 0x0000
```

```
5D4  3A000B     bra nz, 0x0005ec
5D6  E0001E     cp0.w [0x001c]
5D8  320009     bra z, 0x0005ec
                      fractVal += 1.0;
5DA  90005E     mov.w [0x001c+10],0x0000
5DC  9000EE     mov.w [0x001c+12],0x0002
5DE  07FDD4     rcall 0x000188
5E0  200002     mov.w #0x0,0x0004
5E2  23F803     mov.w #0x3f80,0x0006
5E4  07FE9E     rcall 0x000322
5E6  07FE39     rcall 0x00025a
5E8  980750     mov.w 0x0000,[0x001c+10]
5EA  980761     mov.w 0x0002,[0x001c+12]
                   }
                   if (fractVal >= scale) {
5EC  90015E     mov.w [0x001c+10],0x0004
5EE  9001EE     mov.w [0x001c+12],0x0006
5F0  90007E     mov.w [0x001c+14],0x0000
5F2  90088E     mov.w [0x001c+16],0x0002
5F4  510F80     sub.w 0x0004,0x0000,[0x001e]
5F6  598F81     subb.w 0x0006,0x0002,[0x001e]
5F8  350070     bra lts, 0x0006da
                      fractVal = scale - 1.0;
5FA  90007E     mov.w [0x001c+14],0x0000
5FC  90088E     mov.w [0x001c+16],0x0002
5FE  07FDC4     rcall 0x000188
600  200002     mov.w #0x0,0x0004
602  23F803     mov.w #0x3f80,0x0006
604  07FE8D     rcall 0x000320
606  07FE29     rcall 0x00025a
608  980750     mov.w 0x0000,[0x001c+10]
60A  980761     mov.w 0x0002,[0x001c+12]
60C  370066     bra 0x0006da
```

```
                      }
                   } else {/* aVal < 0 */
                      fractVal = ceil (decimalVal);
60E  90003E        mov.w [0x001c+6],0x0000
610  9000CE        mov.w [0x001c+8],0x0002
612  07FEE6        rcall 0x0003e0
614  07FE22        rcall 0x00025a
616  980750        mov.w 0x0000,[0x001c+10]
618  980761        mov.w 0x0002,[0x001c+12]
                      if (fractVal != decimalVal) {
61A  90005E        mov.w [0x001c+10],0x0000
61C  9000EE        mov.w [0x001c+12],0x0002
61E  07FDB4        rcall 0x000188
620  90013E        mov.w [0x001c+6],0x0004
622  9001CE        mov.w [0x001c+8],0x0006
624  07FEE5        rcall 0x0003f0
626  E00000        cp0.w 0x0000
628  3A0001        bra nz, 0x00062c
62A  370009        bra 0x00063e
                         fractVal -= 1.0;
62C  90005E        mov.w [0x001c+10],0x0000
62E  9000EE        mov.w [0x001c+12],0x0002
630  07FDAB        rcall 0x000188
632  200002        mov.w #0x0,0x0004
634  23F803        mov.w #0x3f80,0x0006
636  07FE74        rcall 0x000320
638  07FE10        rcall 0x00025a
63A  980750        mov.w 0x0000,[0x001c+10]
63C  980761        mov.w 0x0002,[0x001c+12]
                      }
                      dummy = fractVal/2.0;
63E  90005E        mov.w [0x001c+10],0x0000
640  9000EE        mov.w [0x001c+12],0x0002
```

```
   642  07FDA2     rcall 0x000188
   644  200002     mov.w #0x0,0x0004
   646  240003     mov.w #0x4000,0x0006
   648  07FE29     rcall 0x00029c
   64A  980710     mov.w 0x0000,[0x001c+2]
   64C  980721     mov.w 0x0002,[0x001c+4]
                      isOdd = (int) ((dummy - floor
(dummy))*2.0);
   64E  90041E     mov.w [0x001c+2],0x0010
   650  9004AE     mov.w [0x001c+4],0x0012
   652  90001E     mov.w [0x001c+2],0x0000
   654  9000AE     mov.w [0x001c+4],0x0002
   656  07FDF9     rcall 0x00024a
   658  BE0100     mov.d 0x0000,0x0004
   65A  BE0008     mov.d 0x0010,0x0000
   65C  07FE61     rcall 0x000320
   65E  BE0100     mov.d 0x0000,0x0004
   660  07FE60     rcall 0x000322
   662  07FDFB     rcall 0x00025a
   664  780F00     mov.w 0x0000,[0x001c]
                      dummy = decimalVal -fractVal;
   666  90005E     mov.w [0x001c+10],0x0000
   668  9000EE     mov.w [0x001c+12],0x0002
   66A  07FD8E     rcall 0x000188
   66C  BE0100     mov.d 0x0000,0x0004
   66E  90003E     mov.w [0x001c+6],0x0000
   670  9000CE     mov.w [0x001c+8],0x0002
   672  07FE56     rcall 0x000320
   674  980710     mov.w 0x0000,[0x001c+2]
   676  980721     mov.w 0x0002,[0x001c+4]
                      if ((dummy > 0.5) || ((dummy == 0.5) &&
isOdd)) {
   678  200002     mov.w #0x0,0x0004
```

```
67A  23F003     mov.w #0x3f00,0x0006
67C  90001E     mov.w [0x001c+2],0x0000
67E  9000AE     mov.w [0x001c+4],0x0002
680  07FDE2     rcall 0x000246
682  E00000     cp0.w 0x0000
684  3C0012     bra gts, 0x0006aa
686  EB4000     clr.b 0x0000
688  985760     mov.b 0x0000,[0x001c+22]
68A  200002     mov.w #0x0,0x0004
68C  23F003     mov.w #0x3f00,0x0006
68E  90001E     mov.w [0x001c+2],0x0000
690  9000AE     mov.w [0x001c+4],0x0002
692  07FEA4     rcall 0x0003dc
694  E00000     cp0.w 0x0000
696  320001     bra z, 0x00069a
698  370002     bra 0x00069e
69A  B3C010     mov.b #0x1,0x0000
69C  985760     mov.b 0x0000,[0x001c+22]
69E  90506E     mov.b [0x001c+22],0x0000
6A0  A20400     btg 0x0000,#0
6A2  E00400     cp0.b 0x0000
6A4  3A000B     bra nz, 0x0006bc
6A6  E0001E     cp0.w [0x001c]
6A8  320009     bra z, 0x0006bc
                      fractVal += 1.0;
6AA  90005E     mov.w [0x001c+10],0x0000
6AC  9000EE     mov.w [0x001c+12],0x0002
6AE  07FD6C     rcall 0x000188
6B0  200002     mov.w #0x0,0x0004
6B2  23F803     mov.w #0x3f80,0x0006
6B4  07FE36     rcall 0x000322
6B6  07FDD1     rcall 0x00025a
6B8  980750     mov.w 0x0000,[0x001c+10]
```

```
6BA  980761     mov.w 0x0002,[0x001c+12]
                   }
                   if (fractVal < -scale) {
6BC  90007E     mov.w [0x001c+14],0x0000
6BE  90088E     mov.w [0x001c+16],0x0002
6C0  100160     subr.w 0x0000,#0,0x0004
6C2  1881E0     subbr.w 0x0002,#0,0x0006
6C4  90005E     mov.w [0x001c+10],0x0000
6C6  9000EE     mov.w [0x001c+12],0x0002
6C8  510F80     sub.w 0x0004,0x0000,[0x001e]
6CA  598F81     subb.w 0x0006,0x0002,[0x001e]
6CC  340006     bra les, 0x0006da
                      fractVal = -scale;
6CE  90007E     mov.w [0x001c+14],0x0000
6D0  90088E     mov.w [0x001c+16],0x0002
6D2  100060     subr.w 0x0000,#0,0x0000
6D4  1880E0     subbr.w 0x0002,#0,0x0002
6D6  980750     mov.w 0x0000,[0x001c+10]
6D8  980761     mov.w 0x0002,[0x001c+12]
                   }
                }

                /* Return fractional value. */
                return ((fractional) fractVal);
6DA  90005E     mov.w [0x001c+10],0x0000

             } /* end of MatrixInverse */
6DC  BE044F     mov.d [--0x001e],0x0010
6DE  FA8000     ulnk
6E0  060000     return
```