Martin Fröjd, Adiel Khan, Jussi Mäkelä

Version: 1.0

CONSTRAINING THE REAL PROBLEM OF FLOATING POINT NUMBER DISTRIBUTION

Abstract- There are many applications that require dealing with real and floating point numbers in electronics ranging from the low-level schematics of analog mixed signal domain to the high-end graphics powering so many personal electronic devices.

One cannot treat floats or reals the same as integers due to the underlying subtleties of the number types. The challenge is developing techniques for enabling verification engineers to correctly stimulate designs with real world scenarios. (IEEE, SystemVerilog Unified Hardware Design, and Verification Language, 1800-2012)(SV) defines a real number as represented by 64bit floating point number see (IEEE, Standard for Binary Floating Point Arithmetic, 754-2008)(IEE754) . The SV language does not address randomization of real or float types. Languages or libraries that do allow randomization of floats are often inefficient and poor distribution of the floating point number line. Whilst there are formal verification techniques better suited for verification of floats, engineers are still required to verify scenarios over multiple levels using dynamic simulation with random pattern injection.

With GPU verification the qualification process requires constrained random pattern stimulus to ensure the color, shape, size and position of graphics primitives hit graphical API, architectural and mathematical corner cases. Numerous float formats are employed by GPU designs for optimization purposes and these need generated during verification. Of those many float formats, a range of values need to be randomly generated to hit interesting corner cases, such as correctly place a graphic primitive in a specific position within frame; dimensions of a graphic object or primitive; perspective / geometric translations, lighting / depth / blending operations and so forth. In such cases randomly generating one floating point number is not useful, a full set of integers with floating point numbers need to be constrained and generated together to ensure correct behavior.

This paper applies the theorems of (Downey, 2007) using SV to highlight differences between good and bad distributions. The implementation must be efficient in its distribution including NaNs, infinities, denormals, zeros, min and max legal ranges whilst being efficient in performance. Also discussed is enabling a user interface similar to integers for constraining floats. Therein, the paper will delve in constraints of floats to using the standard mathematic symbols such as equals, summations, greater-than, less-than, multiplication and divisor.

The result of this effort was base classes in SV that can be used with any floating point number format. The code leveraged ARM's internal C++ Floating point mathematical library package for computations. Then by optimizing the algorithms within the EDA tools used this gave at least 5X gain in dynamic digital simulation runtime performance.

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

I. INTRODUCTION

To open this paper the first question must be, why constrain the generation of floating point numbers and why do you require executing in dynamic digital simulation.

The application under verification is more than what a human will see with their naked eye. In previous GPU iterations a large amount of effort was spent on insuring the visual correctness. A missing pixel at the edge of the screen, or a missing hidden image underneath the current image maybe have been acceptable as negligible to be noticed by the human eye. With the advent of OpenCL framework for GPU's these negligible issues now have significant meaning as the same code that would produce an incorrect pixel or image is now being used to execute compute functionality. Incorrect compute functionality can have disastrous consequences for applications running on the GPU. Floating point numbers are crucial to the operation of the GPU as it uses floats for not just placement of images on a screen but also for defining screen sizes and scissor box areas. Therefore, to accurately verify the GPU system we need to operate in its supported type formats.

Some corner cases are just easier verified using traditional dynamic verification techniques. Historically hard mathematical problems were solved using higher order algorithm tools like HOL. Today one can write C algorithms to perform equivalency checking with RTL. Also, one can write low-level proofs to verify using formal verification tools. In the end each of these has their own pros and cons. HOL like tools require a lot of expertise and manipulation to use correctly. Similarly, C to RTL equivalency checking is great in the presence of C-models for everything but what about glue-logic and cross-over points between the algorithms. Same for formal proofs, without a good complete model the results will be uncertain as to what you are really proving. Finally, if you want to run end-user specific test patterns and target corner-cases within your design then you must have a mechanism within dynamic simulation to execute the scenarios.

The problem is not unique to GPU verification it is also an issue for analog/digital mixed signal (AMS) verification and for HW implemented digital signal processor (DSP) algorithms. For these designs real or float types are used heavily and the dynamic digital simulators with their integer arithmetic API's fall short in providing a usage model for non-integer based verification.

Using SV constraints to target the corner-cases situations within a GPU gives rise to particular inefficiencies within simulators. Longer term these inefficiencies need to be addressed by language and tool experts. Whilst, in AMS the focus can be on IEEE754 real types for GPU/CPU/DSP designs there will often formats beyond the IEEE754 definitions. To that end the language and tools need to provide a solution for a generic floating point format and not just IEEE754.

II. NUMBER FORMATS

Integer numbers are regular in their distance on the number line, whereas floating point numbers are comprised of an exponential factor which results in logarithmic points on the number line. All floating point numbers have three basic elements built within them, the first is a sign value to indicate positive or negative numbers secondly there is an exponent portion to indicate the number of times the base number is multiplied by itself then thirdly there is a mantissa value to represent the significant digits

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

of the number. IEEE754 defines three binary base2 formats with length 32bit, 64bit, 128bit and two decimal base10 formats with length 64bit and 128bit (see table 3.2 of IEEE754). The calculation of the real number from these floating point number formats is calculated as follows:

$$(-1)^{sign} * base^{exponent} * mantissa$$

The format defined in IEEE754 for a 64bit floating point number also referred to as a *double* which has 1bit for sign with 11bits for exponent and 52bits for the mantissa. To understand the nature of how such a number appears on the number line the exponent portion can be manipulated. Increasing the exponent by a value of one will result in the increments of the mantissa to give much larger real number delta on the number line which are characteristic of a logarithmic scale. Later you will see why randomization of a floating point number needs to be controlled otherwise meaningless values can be generated. The range of the exponent is biased to allow for positive and negative exponents, this is controlled by standard bias calculation: $2^{11-1} - 1$

Whilst floating point formats allow for the generation of very small and very large numbers using biased exponent there is a limit to the possible numbers to be represented. To cater for the minimum and maximum limits of the number the exponent when filled with all ones and mantissa is all zeros indicates that the number is representing infinity, where the sign bit dictates if it is a negative or positive infinity value. If the exponent is all ones and mantissa is non-zero then this indicates values which are not numbers such as square root of negative values or deivide by zero values.

Similar issues exists that lead to limitations in the precision of any single number which is defined by the number of bits available for use as the mantissa. Therefore, there are well defined rounding modes to cater for situations when arithmetic calculations require a result beyond the precision available with the mantissa. All floating point numbers are written as scientifically normalized numbers meaning there is only one value before the point. For binary numbers this means that there will always be a 1 before the point, as a 0 would cause point to move right and increase exponent. Thus follows a dilemma as how to represent zero numbers using floating point. The mechanism is appreciating these are denormalized numbers which can be recognized when exponent has all zero bits. In such cases the real number calculation take into consideration that the exponent is zero and assumes 2^{1-bias} rather than the normalized format: $2^{exponent_value-bias}$.

SV defines real and shortreal as 64bit floating point (double float) and 32bit floating point (single float) numbers respectively. The language does not give users a structural representation of a 64bit or 32bit float, nor does is define rounding mode selections. VHDL has defined the fixed and floating point packages allowing users and vendors to know the basic structures present in the design for use with advanced technologies and optimizations. Even if the designs only needed to handle predefined float types the SV language does not define how to perform randomization or coverage of the real and shortreal types. In some respects this is actually quite good as you will see later the randomization has to be thought through intelligently to ensure good distributions. What would be totally inadequate would be to bring in a real number analog style solver from existing analog tools and use that solver in isolation within

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

a digital environment. Many existing real number solvers are targeted to the continuous time analog domain where sweep functions or Monte Carlo analysis is applicable. In the digital simulation domain the applications for randomization of even just the real and shortreal types defined in SV go beyond what is required from these traditional analog solvers. Involving integer calculations with floating point calculations is a key requirement otherwise the verification will not meet the demands for targeting specific areas with sufficient performance. Then going beyond the scope of the real and shortreal types any floating point format should be possible to be used. Of course the normal types are quarter, half, single, double and quads which equate to 8bit, 16bit, 32bit, 64bit and 128bit floating point numbers. It has been common practice in the industry to use internal floating point formats with different mantissa and exponent bit widths for design optimizations and these also need to be handled cleanly by any such solution that a language or tool would be supporting. From the aspect of verifying key components in a GPU design there are so many key features that relate to placement and precision of points on a screen that integers are insufficient and many multiples of floating point numbers are required to correctly draw graphical primitives.

III. CONTEXT OF GPU CORNER CASES

There are many known corner cases scenarios when rendering images onto a display. One such situation that can help with understanding the value of using floating point numbers for verification is the zero-area triangle that actually has a very small area. A triangle where the sum of two sides equals the length of the third will make it a straight-line. There is an important difference between primitive objects like triangles and straight lines. A triangle will have some area associated with it whereas a straight line will only have an associated width.

Classic techniques one learnt to calculate the area of the triangle show us some of the intricacies of verifying with floating point numbers. Heron’s formula for the area of a triangle identifies the zero area triangle issue.

The area of a triangle can be calculated using Heron’s formula:

$$Area = \sqrt{s(s-a)(s-b)(s-c)}$$

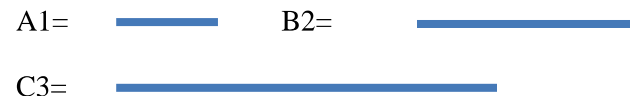
a,b,c are the sides of the triangle, s is the semi-perimeter.

$$s = \frac{a+b+c}{2}$$

Rewritten this becomes:

$$Area = {}^{1}/_{4}\sqrt{4a^2b^2 - (a^2+b^2-c^2)^2}$$

For example, a triangle has sides A1, B2, C3 with values 1, 2, 3.



The only way to ensure that C3 is connected to A1 and B2 is if the angle between A1 and B2 is zero.

In reality side A1 might be greater than 1 by a tiny fractional amount. Hence, it needs to be treated as a triangle with an area and not a straight line. The area could still be too small to be calculated by software models as there is a limit to the maximum precision within which software models can operate. Another

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

way to think of this is that drawing the points of the triangle on a graph, if the Y coordinate is a very large number it will have a finite maximum precision as there are only so many bits in a floating point number before rounding occurs. With a very large Y coordinate the X coordinates between the two other points of the triangle could be less than the precision of Y coordinate. In such cases verification strategies that use software models are often ineffective as rounding modes could give a zero area result.

Interestingly there is an opposite corner case when C3 is a tiny fraction greater than 3 it means that the triangle lines would not meet. In floating point number terminology this is a NaN triangle. A basic triangle rule is that no side shall be greater than the sum of the other two sides. For verification this is an immense challenge as there is no single number, integer or floating point that can cause Zero or NaN triangles. A bug if any could be in any permutation of the 3 sides. By using software models and formal models we can ensure that the RTL is correct to the rounding precisions defined by the models but beyond that there needs to be assurance that the occurrence of NaN and Zero triangles are treated correctly by the design.

Understanding the context from the zero area triangle is important and when placed into the overall context of a drawing mechanism it becomes apparent at the numerous float numbers required to be generated and constrained together.

Prior to drawing any primitives there needs to be a screen and this requires randomizing of the screen resolution. Constraints of the screen need to be weighted for typical resolutions such as 1pixel by 1pixel, 65K by 65k, 1080p, 4K HD. Once a screen is defined there then can be drawn a scissor box within the screen to define where subsequent drawing should occur. The scissor box can also be placed anywhere in the screen so there will need to be constraints to relate the screen and the scissor box locations. Within the scissor box is where primitive shapes are drawn. Each primitive is defined by the vertices (vtx) it uses for locations. A point or dot is a one vertex primitive whereas a line has 2 vertices and a triangle has 3 vertices. Each of these primitives have a variety of attributes for instance a point will have a size and a line will have a width. Each primitive is slightly more advanced than the previous, point =1vtx, line=2 vtx, triangle=3vtx and beyond this are triangle fans and triangle strips which have 3vtx to Nvtx. A triangle fan is a number of triangles joined by the first vertex and the first vertex being at the centre of the fan. A triangle strip is built from triangles joined by vertices 2&3 from previous triangle attaching to vertices 1&2 of current triangle. Now one starts to grasp the level of constraints required between the various stages in drawing primitives. The user must be able to constrain a particular triangle fan to overlap or not overlap with a triangle strip within a scissor box within a screen. Possibly one of the triangles in the fan is a zero-area triangle with a tiny area too small to calculate. Floating point mathematics needs to be used for sizes, positions, areas, lengths, dimensions, along with perspectives overlaid with graphics properties of blending, lighting etc. The verification engineer needs to focus on ensuring scenarios beyond the software models capabilities are verified. These can be in very small or very large floating point number calculations as different calculation techniques are used for design optimizations. An example of an

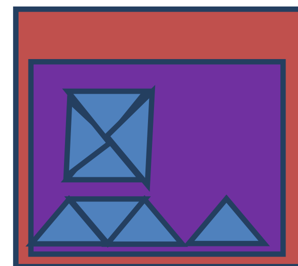


Figure 1

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

optimization you may have done yourself is if a triangle has a 90° angle then the area is simply:

$$A = \frac{1}{2}bh$$

One can mentally compute this much faster than Herons formula however what if the angle is a very tiny fraction greater than 90° or tiny fraction less than 90°. Such interesting crossover points are critical for the verification engineer. Hence this explains the importance of using floating point numbers for as much as possible of the verification challenge.

IV. ADDRESSING DISTRIBUTIONS

A. *Poor distribution vs good distributions*

The task of randomizing floating point numbers is not as straightforward as randomizing integers. Unlike integers – where the values are uniformly distributed over the variable's legal range – the distribution of floating point values is exponential. Additionally, standard floating point number formats contain a number of bit vector representations that encode values outside the number range, such as, 'not a number' (NaN), infinity (+/- INF), zero (+/- zero) and denormals/subnormals.

If a simple approach for randomization of a floating point number is taken where a bit vector is randomized, the encoding of the number format makes it difficult to constrain the vector to fall between useful ranges. Even if constraints are constructed to facilitate randomization of values in a target range, a useful distribution over that range can be difficult due to the exponential distribution, described above. Discounting the exponential distribution of the number format can result in a distribution as shown in Figure 3. To get a more uniform distribution, additional constraints are needed.

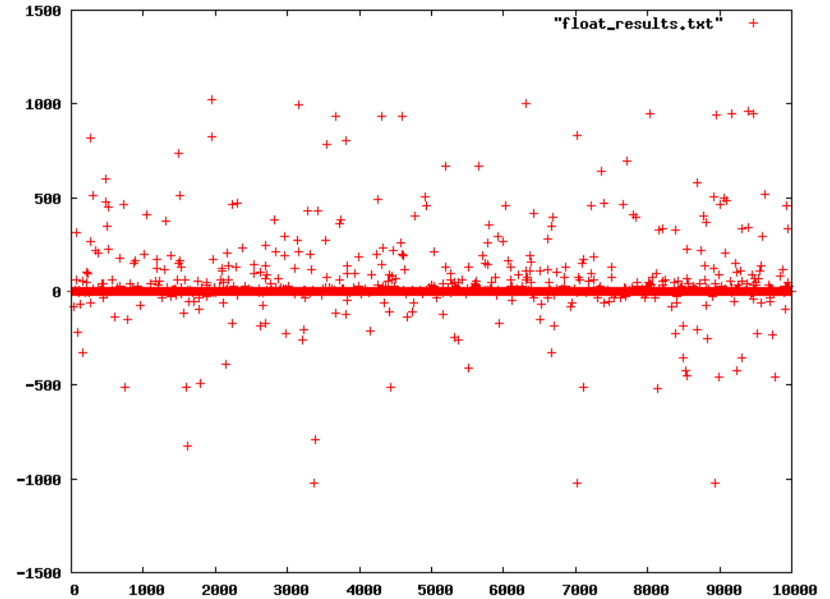


Figure 3

(Downey, 2007) suggested in his paper an algorithm with which a uniform distribution of random values of floating point numbers could be obtained.

The solution described within this paper simplifies the approach that (Downey, 2007) presented to allow the essence of the algorithm to be implemented in the SV language. The solution suggested here works on the premise that floating point values constructed from a binary representation have a 50% probability to be in the highest exponent range. Based on this statement, the following algorithm has been used to produce a uniform distribution when randomizing floating point values:

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

1. Randomize a bit vector which has same width as the exponent. Each bit should have 50% probability to be 1.

2. Loop through randomized bit vector to find the first bit set to 1. The index of the first bit set to 1 defines the value of the exponent, such that the first bit results in a maximum exponent value, and the last bit results in the lowest exponent value.

3. The mantissa is chosen freely except in cases where the randomized exponent value is the same as one or both of the exponent values defining the legal range of randomization. In this case the randomization of the mantissa needs to ensure that the legal range will not be exceeded.

Note that the algorithm presented by (Downey, 2007) also describes an adjustment made to exponent based on the value of the mantissa to correct the distribution for the first point in the range. However, for simplicity that step was not implemented in solution presented here.

The practical solution to capture constraints and other functionality (such as arithmetic operations) in re-usable way is to implement a class to encapsulate the floating point number.

The challenge in implementing the suggested algorithm using the features of the SV is to get such an implementation with which several floating point numbers can be solved during the randomization phase such that one random floating point value can be used as a range limit to another. This requirement limits the way ranges can be set, and also what parts of randomization or the algorithm can be placed in the pre_randomize() and post_randomize() functions.

The structure of the algorithm suggests that it would be easiest to place minimal constraints in the randomization phase and execute most of the value selection in the post randomize phase. However, this is not possible due to the sequence and relationship of the SV randomization behavior. For example; when classes are randomized the pre_randomize() functions in all random classes are executed. Following the completion of the pre_randomisation phase the actual randomization will be executed together, and finally the post_randomize() functions in all classes will be executed. With this relationship, the random floating point value produced in one instance has a dependency on the on another yet the randomization of these two instances are in lock-step, which will result in an invalid result.

The solution proposed for resolving randomization dependency for setting limits such that they are solved during the randomization phase is to add class member variables for upper and lower limits and define them as rand variables. This resolves the dependency and allows randomizing float classes so that one float can act as a limit to another.

```
class numbers_float_generic ;
...
  rand packed_float_struct_t upper_limit;
  rand packed_float_struct_t lower_limit;
  rand packed_float_struct_t value;
...
endclass
```

To simplify the implementation of the algorithm, all constraints for the exponent selection can be implemented in the

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

pre_randomize() phase so that there is no need to know the limits. This can be done such that a bit vector with a width of exponent is randomized and the position of first bit set to 1 in the vector is set as value to be subtracted from the largest exponent. Subtraction itself will be placed into the constraints.

```
function void pre_randomize();
  int unsigned exponent_chooser_vec;
  exponent_chooser_vec=
             $urandom_range({EXPONENT_WIDTH{1'b1}}, 0);

//Select max value in case exponent_chooser_vec is 0.
  exponent_chooser = EXPONENT_WIDTH;

  for (int i = 0; i < EXPONENT_WIDTH; i++) begin
    if (((exponent_chooser_vec >> i) & 'h1) == 1) begin
      exponent_chooser = i;
      break;
    end
  end
endfunction

...
constraint exponent {
 if (upper_limit.exponent >= exponent_chooser) {
  value.exponent==upper_limit.exponent-exponent_chooser;
 }
 else { value.exponent == 0; }
}
```

The implemented solution gives a much improved distribution for random floating point number over range which is equally distributed on both sides of 0. This can be seen in Figure 4, which show values of 10,000 randomized floating point numbers within range -1023..1023.

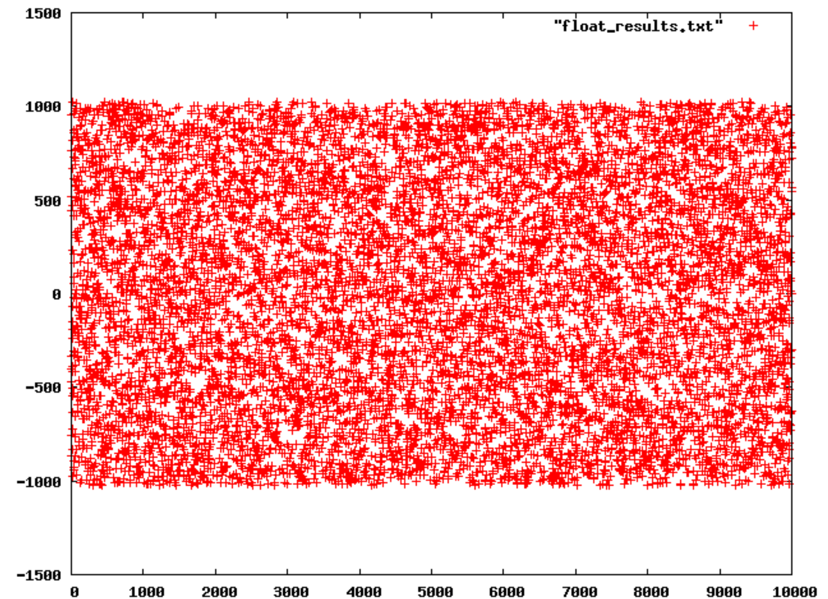


Figure 4

B. *Setting ranges non-balance over 0*

In cases where the range crosses zero but the range isn't centered around zero, the algorithm proposed so far will result in an uneven distribution as it still provides a 50:50 split for positive and negative values. Figure 5 shows an example of this where the negative range is smaller than the positive range and as a result a higher density of hits.

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

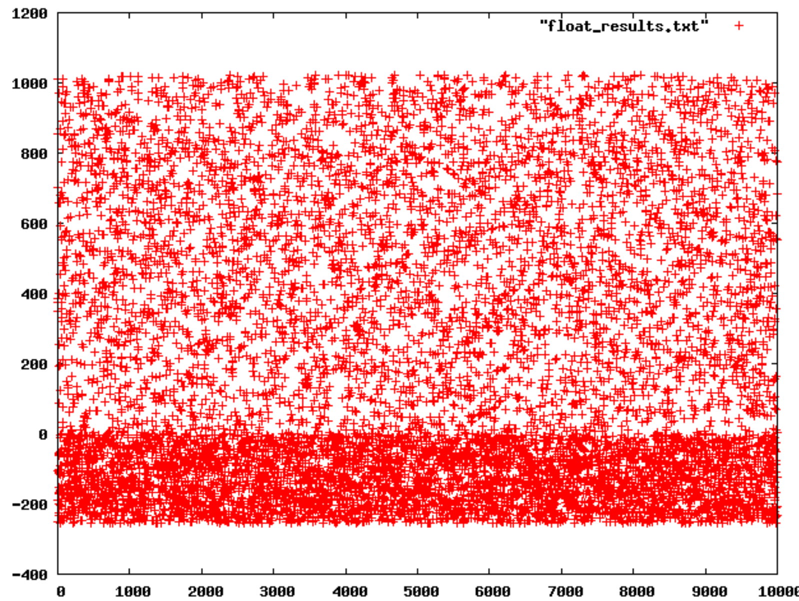


Figure 5

A practical fix to this issue is still to use original algorithm to choose the exponent, but also to add a constraint on the largest exponent chosen based on the sign bit. To balance the proportion of hits related to size of the range, the difference in the exponent values has been used to define size of the ranges on each side and to adjust the distribution of sign.

```
constraint sign {
  if (upper_limit.exponent > lower_limit.exponent) {
     value.sign dist {
      upper_limit.sign :=
      (upper_limit.exponent - lower_limit.exponent + 1),
       lower_limit.sign := 1
     };
  } else {
  value.sign dist {
     upper_limit.sign := 1,
       lower_limit.sign :=
         (lower_limit.exponent - upper_limit.exponent + 1)
       };
    }
}
```

In Figure 6, the affect of sign distribution change can be seen.

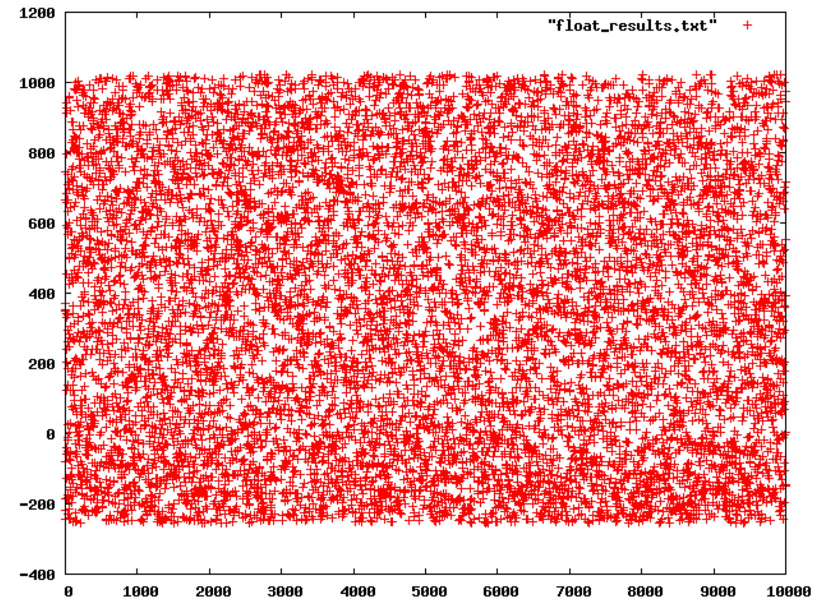


Figure 6

C. *Non-full range of mantissa per top/bottom exp*

As shown so far, the proposed solution gives a good distribution of values through a workable constraint mechanism. However, the solution contains a flaw which has not been resolved.

The current solution assumes that whole mantissa range will be used. However, if the limit imposed by the user constraint does

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

not allow whole mantissa range to be used, the largest exponent range will still be chosen in 50% of cases (as result of exponent selection presented above). In this case, there will be larger density of hits on largest exponent range. The affect of this is demonstrated in Figure 7 which shows the results of randomizing floating point values between 0 and 1024. The limit value of 1024 is on a new exponent range (137) but limits the mantissa to one possible value (0) and so value of 1024 gets 50% of hits.

This issue was not solved since it was decided that if users are aware of this it can be avoided and even if not avoided the solution gives good amount of hits on the remaining range.

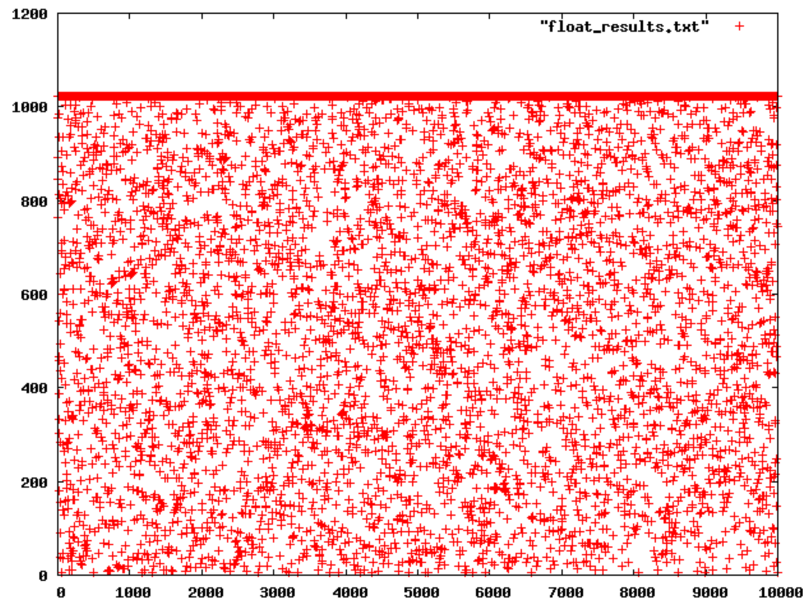


Figure 7

D. *Coarse Ranges*

Even though values which do not present any number and infinity might not have a use case in floating point arithmetic they might be valid test cases in hardware verification. For example it might be requirement that unit correctly discards invalid or not supported values, so in this case it is critical that these values can be generated.

Also values like 0, smallest non-zero value, largest possible value etc. might become rare when values in floating point range are uniformly distributed. In floating point arithmetic these are part of interesting cases and there should be possibility to increase probability to hit these values.

To provide the functionality for the user to affect probability of these values, a separate field to select coarse range was added. Coarse range in this case is a random enumerated value which can be used to select specific value, value range or uniform distribution.

E. *Usability and Human Readability*

The constraints and functionality presented in previous sections achieve the aim to get uniformly distributed random floating point numbers. However, for a reusable solution, the ease of use and human readability of constraints cannot be ignored.

The main factor affecting usability is the conversion from the number value to the bit vector representation of a floating point number. As presented in section II, arithmetic operations must be applied to the bit vector representation to obtain the floating point number value. To avoid the need to have knowledge and/or execute these operations when setting or reading constraints in

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

test cases, the user of the floating point class library should be able to set values appropriate to the scenario. In most cases this will be as a floating point value, in which case the system should deal with the conversion using the require operations. Setting values in this way also enables better portability of the test cases.

To achieve usability, human readability and format independent value setting in test cases, strings were used to define floating point values in the constraints and string-to-float functions were used to convert the string to a bit vector. For simplicity the conversion itself was implemented in C. The goal of readability and usability was achieved with this approach, but at a cost of a drop in performance during randomization due to an increase in simulation DPI calls.

Performance penalties were noted and different ways to convert number value to float were considered but not implemented.

F. *Practical Issues*

During deployment in real applications, several issues were encountered which eventually limited the use of the floating point class library that raised the need for an improved solution. These issues are partially due to the implementation and partially due to the functionality of the SystemVerilog.

The major limiting factor was the constraint solver and complexity of constraints. The use of rand fields for setting the constraint ranges provided a working solution. However, this solution can cause problems for the constraint solver in cases where there is a large chain of constraints with several floating point numbers used to limit to each other. In such cases, the constraint matrix is complex and the problem hit limits of the constraint solver which manifest as performance degradation, higher memory usage requirements and contradiction errors. The performance of the constraint solver could have been improved if further guidance related to the implemented algorithm could have been given to the tool.

The secondary limiting factors to performance related to the solutions implemented to improve usability, as described in section E, and the overhead incurred in the iterative solution required to produce randomly sized arrays of floating point numbers. Depending on the use case for the randomly sized array, there are multiple options. For example, the array size can be randomized and necessary amount of classes constructed in pre_randomize() function. Alternatively a maximum amount of classes can be constructed in the pre_randomize() function and the array size is cut to correct size in the post_randomize(). If the size of the array is randomized in pre_randomize() function it limits what constraints can later be added for size. Creating maximum number of classes has negative affect to performance and causes unnecessarily large memory consumption.

The final limiting factor of note relates to the usability of the solution. If the scenario requires several random floating point numbers with interdependencies, it was found to be very sensitive to constraint changes. Thus refining constraints towards a target scenario burnt additional user effort.

V. CONCLUSION

The SV base classes were used by all of the modular level testbenches and on the system-level testbenches. With constructive feedback on the usability and the performance of the code the implementation went through an iterative improvement process. One of the key points for end users of the floating point

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.

library to understand was that the digital simulators handle normal integer constraints much better than the more complex constraints built into the floating point library. This meant that with extensive whitebox knowledge of the designs under verification the engineers could ensure that for the majority of the time normal integers were used and only when necessary was the floating point library employed.

Due to the feedback received the library improved its performance dramatically but at some point there has to be a tradeoff between required features and usability. Much of the lack of performance was from how constraint solver engines within digital simulators were dealing with the libraries constructs. By explaining the nature of the library also explaining how the constraints were being modeled for improved distributions the solver engineering team was able to recognize redundant implications and dependencies within the constraints solution space. Along with other improvements the simulator was able to provide a 5X gain in performance on a real design which heavily used the floating point library.

To ensure algorithm compatibility with software models the library uses C code functions for many of the arithmetic operations such as addition, multiplication, divide and multiply. For a more native solution it would be appropriate to perform all off these functions within SV code. This could also remove the heavy burden of C code performing string to float manipulation just so the users can write "1.5" style syntax in SV code.

The library as it stands solves many of the verification issues but it has flaws as mentioned previously, such has not scaling to partial mantissa ranges. As and when future projects require enhancements to the library these will be implemented to address ever increasing users.

It is the intention and hope of the authors that the industry will recognize the works written here as necessary to be implemented natively within tools and languages used for verification of digital blocks that deal with real or floating point number types.

VI. ACKNOWLEDGEMENTS

The authors would like to acknowledge key contributions from Robin Hotchkiss of ARM and Katherine Qiang of Synopsys. Without the support and insight from R. Hotchkiss this problem may have burdened ARM engineers for longer than necessary. We owe a great many thanks to K. Qiang for her innovation in creating new algorithms to ensure the code developed by ARM is sustainable and usable with sufficient performance from the simulator toolset. An extended thank you goes to all members of the MALI GPU team whom have tested and gave constructive feedback on the code in use.

VII. REFERENCES

Downey, A. (2007, July 25). Generating Pseudo-random Floating-Point Values.

IEEE. (754-2008). Standard for Binary Floating Point Arithmetic. *IEEE* . IEEE CS.

IEEE. (1800-2012). SystemVerilog Unified Hardware Design, and Verification Language. *IEEE Standard for SystemVerilog* . IEEE SACAG.

http://en.wikipedia.org/wiki/OpenCL

http://www.glprogramming.com

http://malideveloper.arm.com

Copyright 2013-2014 by ARM, Ltd., and Synopsys International Limited. All rights reserved.