2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 15-16, 2024

Addressing Fixed-Point Format Issues in FPGA Prototyping with an Open-Source Framework

Vishal Chovatiya, Infineon Technologies, Bangalore, India (vishal.chovatiya@infineon.com)

Gabriel Rutsch, Infineon Technologies, Neubiberg, Germany (gabriel.rutsch@infineon.com)

Wolfgang Ecker, Infineon Technologies, Neubiberg, Germany (wolfgang.ecker@infineon.com)

***Abstract*—Implementing algorithms on FPGAs is vital for designing, developing, and prototyping algorithms in various fields. FPGAs offer improved performance compared to software implementations and the ability to test algorithms in real-world conditions. However, FPGAs have limited hardware resources and algorithms often use resource-intensive floating-point arithmetic. To address this issue, algorithm designers switch to fixed-point arithmetic, which is faster and requires fewer resources. However, switching to a fixed-point format can lead to range violation and loss of precision. Therefore, it is crucial to validate and verify the correctness of algorithms after conversion to the fixed-point format. Moreover, debugging algorithms that use fixed-point arithmetic during emulation can be challenging, particularly for hardware-in-loop or customer-design-in setups. This paper is an extension of a lightweight, open-source and tool-agnostic framework to simplify the process of algorithm implementation and verification that involves fixed-point datatypes and arithmetic onto FPGA. Additionally, the paper systematically and effectively addresses challenges related to utilizing a fixed-point format, including precision loss, range violations (overflow/underflow), and algorithm debugging during simulation and emulation. The presented method is demonstrated with the effective development of the power conversion circuit algorithm.**

Keywords*—Fixed-point format issues, Fixed point analysis and debugging during FPGA prototyping, SVREAL, Synthesizable real number library in SystemVerilog

I. Introduction

The fixed-point representation, an integer format with an implied binary point (also called radix-point), is an alternative to the IEEE 754 floating-point standard. The fixed-point format is preferred in place of the IEEE 754 floating-point format for reasons like faster arithmetic, lower cost of design development, less power usage for the design, less hardware-software implementation resources, etc. And it fits well into integer data paths, so no additional hardware circuitry is required. Though, porting between floating and fixed-point formats often faces range issues (overflow/underflow) and precision loss. Also, fixed-point variables in hardware description languages (HDLs) are typically represented as bit arrays with a signed integer format. This makes analysis and debugging difficult during simulation and emulation as bit arrays require conversion to real numbers using binary point information.

One such use case of fixed-point format is Analog/Mixed Signal simulation conducted on FPGA [14], where the behavior of analog circuitry is modelled with the SystemVerilog standard-supported *real* data type. Analysis and debugging of the *real* data type is straightforward during the simulation as the language standard natively supports it. However, most synthesizers do not synthesize the *real* type. Therefore, switching to fixed-point format and associated arithmetic is required if an algorithm or design is planned to be emulated, which is necessary and often the case to achieve faster simulation speed and customer-design-ins. Synthesis of the fixed-point format is easy as it stores the real value in a fixed-width multi-bit buffer with an implicit binary point. But analyzing the value stored in this buffer is not straightforward, consequently debugging the algorithm or design being implemented.

II. Related Work

When prototyping algorithms on an FPGA that require real numbers, switching between fixed-point and floating-point formats with no or minimal design change is necessary. It is also essential to have a mechanism to identify range violations, precision loss and real number interpretation from the fixed-point buffer during the simulation and FPGA emulation, ensuring equivalence of fixed-point values against floating point values.

While there are established solutions to work with fixed-point formats, like MathWorks's Fixed-Point Designer [10] and Vitis HLS's `ap_fixed` type [11], these are commercial and tool-dependent solutions that work at a high level of abstraction, limiting HDL-level granularity.

At present, publicly available fixed-point HDL libraries are FPHDL [9] and `Verilog Fixed point math library` [7] by Sam Skalicky. However, FPHDL support is more complex with the Vivado toolchain, and both libraries need a mechanism to detect precision loss issues and real value interpretation mechanisms during simulation and emulation. Moreover, quick port of fixed-point format to floating-point format and vice-versa needs to be included.

The aforementioned limitations have motivated us to develop an extension to a lightweight, robust, open-source framework that combines various practical approaches to overcome the shortcomings of fixed-point format during FPGA prototyping.

III. Framework

Understanding fixed-point variable declaration is important to mitigate various fixed-point format issues. The method demonstrated here is based on the publicly available SVREAL [1] library.

A. *Fixed-point Format Variable Declaration*

A fixed-point number represented as a two's complement signed integer *s* with an implicit exponent *p* and real number *v* i.e.,

$$s \approx v \cdot 2^{p}$$

A 32-bit fixed-point format with a 16-bit for fractional has the following fields:

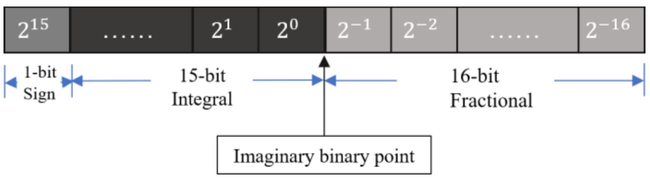


Figure 1: 32-bit fixed-point representation.

The smallest and largest real value with the 32-bit fixed-point format can be represented as:

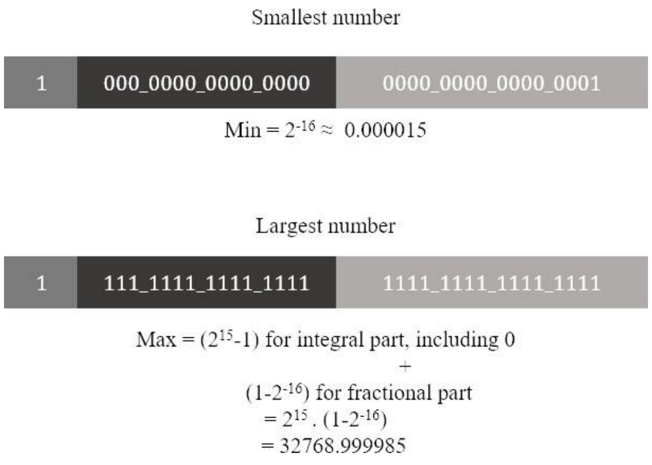


Figure 2: Largest and smallest value with 32-bit fixed-point format.

A declaration of fixed-point format variable in System Verilog is made in listing 1

```
`define WIDTH 32 // fixed point buffer width

`define DATA_TYPE_REAL(width_expr) \
    `ifdef FLOAT_REAL \
        real \
    `else \
        logic signed [((width_expr)-1):0] \
    `endif

`define CALC_EXP(range_expr, width_expr)
    (clog2((real'(range_expr)) \
    / \
    ((2.0**((width_expr)-1))-1.0)))

`define MAKE_REAL(name, range_expr) \
    `DATA_TYPE_REAL(`WIDTH)    ``name``; \
    localparam real ``name``_range_val = range_expr;
        \
    localparam integer ``name``_width_val = `WIDTH; \
    localparam integer ``name``_exponent_val = \
                       `CALC_EXP(range_expr, `WIDTH);


module XYZ;
    ...
    `MAKE_REAL(variable, 10.0);
    ...
endmodule
```

Code Listing 1: Fixed-point format variable declaration

The process of declaring fixed-point format variables involves explicitly specifying their range and width, while exponents are calculated implicitly. The range of a fixed-point variable represents a valid numerical range that spans from negative to positive values. By providing a range, the exponent and alignment details are handled automatically, enabling convenient customization of the format for each variable without any complications. A variable's range controls its resolution; a smaller range provides a finer resolution. Overflow occurs if the range specified during fixed-point variable declaration exceeds the buffer capacity. This can be mitigated by adding an assertion on implicitly calculated exponent value within fixed-point variable declaration macro (i.e. *MAKE_REAL*). Keeping the buffer width as a global macro provides flexibility to configure it according to FPGA DSP (Digital Signal Processing) slice input width, so heavy operations like multiplication will utilize exactly one DSP slice per operation. However, there is still a provision to customize fixed-point variables individually in SVREAL [1].

Moreover, switching between fixed-point and floating-point is easy with just a macro definition, allowing for minimal changes when running algorithms on host PC simulations (with *real* data type) and emulations (with fixed-point data type).

B. *Mitigating Fixed-Point Format Challenges*

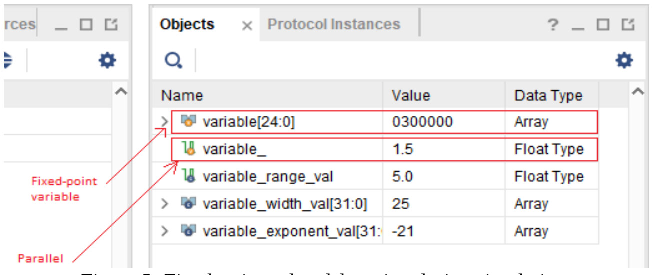
1) In Case of Simulation:

a) Analysis and Debugging Issue: Algorithm development typically uses host PC simulation and the SystemVerilog-supported *real* data type. Converting data types from fixed-point to floating-point is straightforward due to how variables are declared. The *real* data type is simulatable in most simulators, making analyzing and debugging real values during simulation straightforward.

```
`define MAKE_REAL(name, range_expr) \
    `DATA_TYPE_REAL(`WIDTH)    ``name``; \
    localparam real ``name``_range_val = range_expr;
        \
    localparam integer ``name``_width_val = `WIDTH; \
    localparam integer ``name``_exponent_val = \
                        `CALC_EXP(range_expr, `WIDTH); \
    `ifdef FIXED_POINT_SIMULATION_DEBUG \
        real ``name``_; \ // parallel real type
            variable
        always @(name) begin \
            ``name``_ = `TO_REAL(name); \
        end \
    `endif
```

Code Listing 2: Fixed-point format debugging mechanism

However, before switching the data type to a fixed-point format and synthesizing the algorithm, verifying the correctness of the fixed-point values against the floating-point values in the simulation is essential. This can be done by defining a implicit parallel *real* type variable that will be updated simultaneously based on changes in the fixed-point variable, as demonstrated in listing 2.



Name	Value	Data Type
variable[24:0]	0300000	Array
variable_	1.5	Float Type
variable_range_val	5.0	Float Type
variable_width_val[31:0]	25	Array
variable_exponent_val[31:	-21	Array

Figure 3: Fixed-point value debugging during simulation.

Figure 3 shows the fixed-point format variable with a parallel floating-point format variable that helps in interpreting fixed-point format value during simulation:

b) Range Issue: Range issues are basically overflow or underflow conditions which can easily be detected with *real* data type and range assertions during simulation. Range issues can be identified by comparing user-provided (at the time of fixed-point variable declaration) range and operation result or current variable value. This is already been implemented in SVREAL [1] library.

```
`define MAKE_REAL(name, range_expr) \
    `DATA_TYPE_REAL(`WIDTH)    ``name``; \
    localparam real ``name``_range_val = range_expr;
        \
    localparam integer ``name``_width_val = `WIDTH; \
    localparam integer ``name``_exponent_val = \
                            `CALC_EXP(range_expr, `WIDTH)
                                ; \
    `ifdef FIXED_POINT_SIMULATION_DEBUG \
        real ``name``_; \ /*updated by operations*/
        `ifdef COMPARE_FIXED_TO_FLOAT \
            logic ``name``_differ; \
        `endif \
        always @(name) begin \
            `ifdef COMPARE_FIXED_TO_FLOAT \
                ``name``_differ = \
                !`IS_EQUAL_REAL(``name``_, `TO_REAL(
                    name)); \
            `endif \
        end \
    `endif
```

Code Listing 3: Precision loss identification mechanism

c) Precision Loss Issue: Precision errors occur when the number of bits used to represent a value is not enough. This can happen when using a fixed-width buffer to store a real number, as there are infinite real numbers even within a small range like 0.0 to 0.1. These errors can be detected by comparing a shadow real type variable (declared to analyse fixed-point values) with the fixed-point buffer during the simulation in listing 3. This can easily be extended to establish a tolerance level feature that evaluates the fixed-type value against the real-type value within an acceptable range, which is demonstrated in the case study section.

2) In Case of Emulation:

During simulation, the limitations associated with fixed-point formats including range and precision loss can be addressed. However, during emulation, the main challenge is related to the real number value interpretation of the fixed-point buffer. This is because the debugging utilities provided by FPGA vendors, like Integrated Logic Analyzer (ILA [3]) or Virtual Input Output (VIO [4]) for Xilinx, do not support fixed-point representation. These utilities only display digital single or multi-bit logic and convert it to different formats, like signed/unsigned integers, binary or hexadecimal. However, they cannot convert the fixed-point bit stream to a real value due to the absence of binary point information.

```
`define IFA_TRACE(in_name) \
    logic signed [(``in_name``_width_val + \
                   ``in_name``_exponent_val)-1):0]
                       \
                   ``in_name``_integral; \
    logic unsigned [(24-1):0]  ``in_name``_fractional
        ; \
    logic unsigned [(3-1):0]  ``in_name``_exponent; \
    IFA #( \
        .``port``_range_val(``in_name``_range_val), \
        .``port``_width_val(``in_name``_width_val), \
        .``port``_exponent_val(
            ``in_name``_exponent_val) \
    ) ``in_name``_ifa ( \
        .in(in_name),  \
        .integral(``in_name``_integral), \
        .fractional(``in_name``_fractional), \
        .exponent(``in_name``_exponent) \
    );

`define MAKE_REAL(name, range_expr) \
    `DATA_TYPE_REAL(`WIDTH)    ``name``; \
    localparam real ``name``_range_val = range_expr;
        \
    localparam integer ``name``_width_val = `WIDTH; \
    localparam integer ``name``_exponent_val = \
                       `CALC_EXP(range_expr, `WIDTH); \
    `ifdef FIXED_POINT_SIMULATION_DEBUG \
        real ``name``_; \ /*updated by operations*/
        `ifdef COMPARE_FIXED_TO_FLOAT \
            logic ``name``_differ; \
        `endif \
        always @(name) begin \
            `ifdef COMPARE_FIXED_TO_FLOAT \
                ``name``_differ = \
                !`IS_EQUAL_REAL(``name``_, `TO_REAL(
                    name)); \
            `endif \
        end \
    `else \ // Emulation
        `IFA_TRACE(name); \ // IFA module
            instantiation
    `endif
```

Code Listing 4: Fixed-point format debugging mechanism during emulation

An efficient way to interpret real values through a fixed- point buffer is to use FPGA fabric to process the signal in real-time and drive the interpreted value through ILA [3]/VIO [4] interfaces during emulation. The real value is split into integral and fractional parts represented by different integer variables. An additional variable, an exponent, is needed to address leading zeros in the fractional part.

a) Integrated Fixed-point Analyzer(IFA) Architecture: When declaring a fixed-point variable, an IFA module is instantiated, which separates the value into three integer variables (as shown in listing 4). These variables update simultaneously and send their values to the host PC through the supported radix type of ILA [3] or VIO [4].

To understand how IFA processes fixed-point buffers, consider the example of storing -1.5 real value to a 32-bit fixed-point buffer with 16 bits reserved for the fractional part. In that case signed integer s is:

$$s \approx v \cdot 2^p = -1.5 * 2^{16} = -98,304 = (FFFE\_8000)_{16}$$

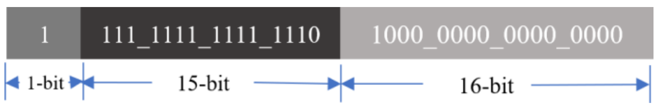


Figure 4: 32-bit fixed-point buffer with '-1.5' real value.

Integral computation: is straightforward since the original 32-bit buffer is split into 16 most significant bits (MSBs), including the sign bit. Negative numbers can be identified by the MSB, the integral part is incremented by 1 to obtain the actual number. Thus, the resultant value 1111_1111_1111_1111 represents −1 in signed integer format.

Fractional computation: is done by converting a 16-bit fractional split buffer to 2's complement if it is a negative number. Then fraction(F) can be computed with:

$$F = \sum_{k=n}^{0} f[k] \cdot 2^{-k}$$

Here, $f[k]$ is k^{th} bit in fractional split buffer, and n is the MSB number.

For our example of 16-bit fractional, computation of decimal fraction is done as:

$$(8000)_{16} = (10...0)_2 = 1\times2^{-1} + ... + 0\times2^{-16} = (0.5)_{10}$$

Here, the challenge is the multiplication operation as it is expensive in FPGA. Hence, the precomputed table of 2's negative power is stored in unsigned integer format. And add up the relative value based on set bits in the fractional buffer. For 16-bit fractional width, the table will have 16 unsigned integer values by discarding decimal points like $2^{-1} = 5000000$, $2^{-2} = 2500000$, . . . , up to $2^{-16} = 0000153$. The limitation here is the pre-computed value buffer will dictate the precision of the fixed-point number. The larger the buffer, the more precise the fraction will be.

For example, if you prepare the precomputed table with a buffer width of 24-bit then the maximum decimal digit number you can represent in the fractional part is $2^{24} - 1 = 16,777,215$. So the full range of decimal digit fractional part can represent is $\lfloor log_{10}(16,777,215) \rfloor \approx 7$ digit i.e. from 000_0000 to 999_9999.

Exponent computation: presents both challenges and simplicity. Initially, one might question the necessity of performing this computation. However, it becomes apparent in scenarios when dealing with a real value like −1.0625. In this case, −1.0625 is represented as an integral part with a value of −1 and a fractional part with 625000, as the fractional part is interpreted as an unsigned integer with leading zeros being disregarded. Therefore, an extra variable is required to represent the number of leading zeros. The calculation of the exponent is directly related to the buffer width of the fractional part. For example, in case of real value −1.0625, the exponent will be 7 (maximum fractional decimal digit with 24-bit fractional buffer width) − (minus) 6 (decimal digit in calculated fractional that 625000) = 1.

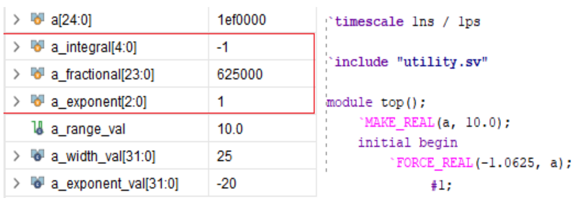
Therefore, overall fraction will be $(10^{-exponent} \,||\, fractional) = (10^{-1} \,||\, 625000)$ interpreted as 0.0625000. And resultant real value of the fixed-point buffer can be interpreted with a bundle of three independent variables, integral, fractional, and exponent, as shown in 5.

$$Integral + (10^{-exponent} \,||\, fractional))$$

For example, −1.0625 will be interpreted as −1 + (0.0 || 625000) = −1.0625000

IV. CASE STUDY

The usage of the presented framework extensions is shown in the example of an algorithmic model of a buck converter circuit as depicted in figure 6. A buck converter is a topology for DC-to-DC conversion, where the voltage of the DC input line is stepped down to a targeted output voltage while stepping up the current.



a[24:0]	1ef0000
a_integral[4:0]	-1
a_fractional[23:0]	625000
a_exponent[2:0]	1
a_range_val	10.0
a_width_val[31:0]	25
a_exponent_val[31:0]	-20

```
`timescale 1ns / 1ps

`include "utility.sv"

module top();
    `MAKE_REAL(a, 10.0);
    initial begin
        `FORCE_REAL(-1.0625, a);
            #1;
```

Figure 5: Fixed-point value interpretation in emulation

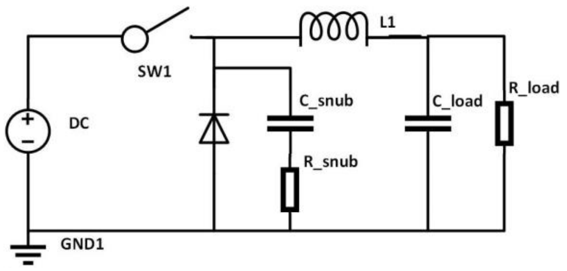


Figure 6: Buck converter circuit schematic

For this case study, we made use of **msdsl** [12] to model the buck converter as a system of linear differential equations (LDS) illustrated in listing 5. This is a common approach to model physical systems [13] and shows the broad applicability of the proposed framework extensions.

```
# IOs
digital_input('hs')
digital_input('ls')
analog_input('v_in')
analog_output('v_out')
# internal state variable
analog_state('v_snub', 5)
# Current into snubber capacitor
i_snub==(v_sw-v_snub)/r_snub
# KCL at switch node
hs/r_sw*(v_in-v_sw)==ls/r_sw*v_sw+i_snub+i_ind
# Inductor behavior
Deriv(i_ind)==(v_sw-v_out)/ind
# Capacitors
Deriv(v_snub)==i_snub/c_snub
Deriv(v_out)==(i_ind-v_out/r_load)/c_load
```

Code Listing 5: Buck converter algorithm excerpt

```
// I/O definition
`MAKE_CONST_REAL(5.0, v_in);
`MAKE_REAL(v_out, 5.0);
`MAKE_REAL(i_ind, 10.0);
`MAKE_REAL_ABS_TOL(i_snub, 10.0, 1.0e-2);

// Diode emulation comparison
`GT_REAL(i_ind, ls_thresh, ls_en);
// High-side & Low-side signal
`PWM(0.50, 500e3, hs);
logic ls;
assign ls = (~hs) & ls_en;
```

Code Listing 6: Testbench stimulus excerpt to buck converter model

The stimulus of the buck converter is depicted in listing 6 and simulation results are shown in figure 7. The original model is tested for an input voltage (v_in) range of +5/-5 volts. In order to analyze a different application, v_in is changed from 5V to 24V. Also, the ranges for v_out, i_ind and i_snub are adapted with the same range ratio and set to 24V and 48A. Running a simulation using floating point data types shows the same results as simulations with the input voltage set to 5V. Using the fixed-point format however shows significant deviation from the expected behavior despite previous range adaptations, see figure 8. For signals v_out, i_snub and i_ind in the corresponding upper graphs the waveforms of the floating-point representation in blue and the fixed-point representation in orange are shown. The corresponding lower graphs show the error between the fixed-point and floating-point representations, specified error tolerances and the tolerance checker signal to indicate tolerance violations.

```
Real number d_aligned with value -7.738281 out of
    range (-5.000000 to 5.000000).
Time: 1730 ns  Iteration: 2  Process: <design path>
    Scope: <design path>.dff_real_v_snub_i.
    assertion_real_d_aligned_i  File: <file path>
```

Code Listing 7: Message of a reported range violation.

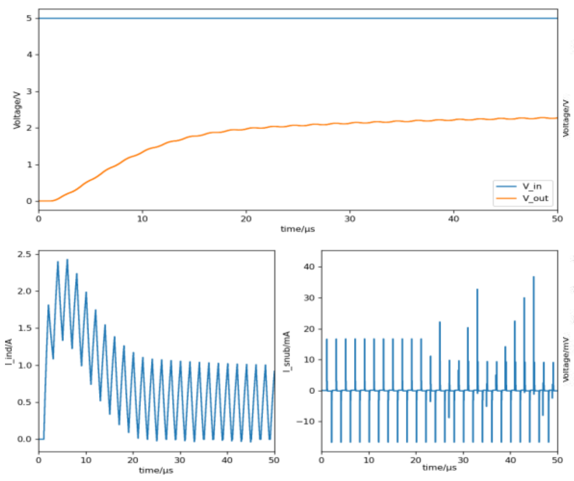


Figure 7: Input and output voltage with switching and leakage current of a buck converter during ramp-up.

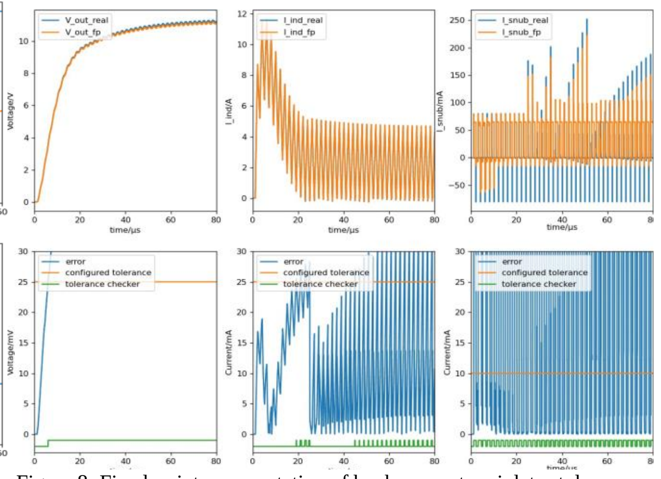


Figure 8: Fixed-point representation of buck converter violates tolerances after changing v_in from 5V to 24V.

To resolve these tolerance violations, the presented framework extensions help to systematically analyze, understand, and address these fixed-point format related effects. First, range assertion violations can be identified by setting the RANGE_ASSERTIONS define when running a simulation. Range assertion violations are reported as displayed in listing 7. It can be seen that the range for the analog state v_snub is not sufficient. Compared to analog signals, ranges for analog states are specified within the model and not in the testbench which makes it easier to miss adapting them. After the identified range violations are fixed, it can already be observed in figure 9, that the deviation in behavior between the floating-point and fixed-point representations has significantly decreased.

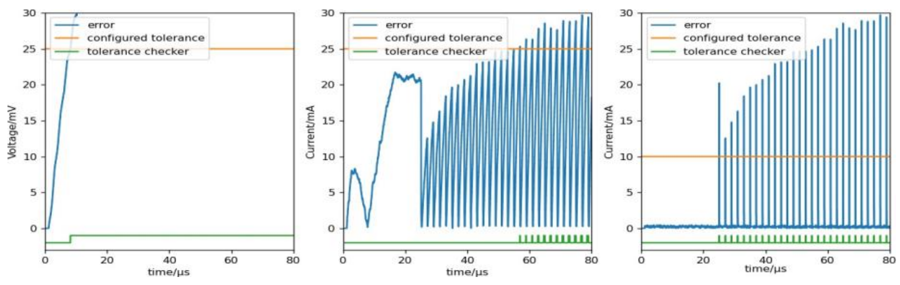


Figure 9: Deviation between fixed-point and floating-point representation has decreased after fixing range violations.

In order to better understand where the remaining precision violations occur, the tolerance checker signals for each variable of the algorithmic model can be traced. Identified precision violations can be fixed through a gradual bit width increase for variables with tolerance violations. In the case of the buck converter model, bit widths of variables related to the computation of i_ ind had to be increased by three to resolve all remaining violations see figure 10.

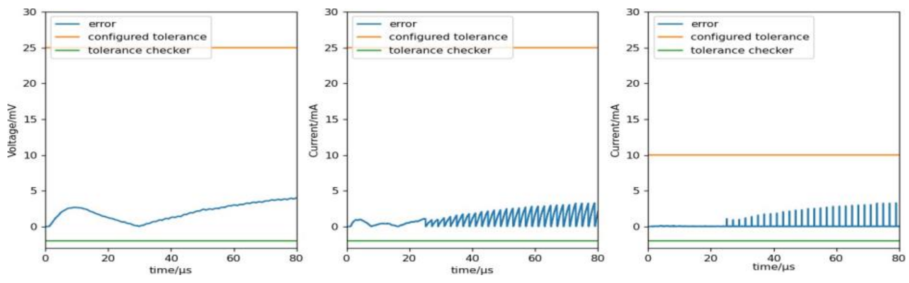


Figure 40: Fixed-point representation of buck converter is no longer violating tolerances after tuning variable precision.

Table 1: Resource Utilization and Timing (Pynq-Z1)

	LUT	FF	BRAM	DSP
Trace Unit w/o IFA	1503	2646	11	0
Trace Unit w IFA	2824	4763	19	0
IFA units	305	0	0	0
Total resource incr.	**2000**	**2117**	**8**	**0**

Timing			
	without IFA	with IFA	Diff.
Clock	25MHz	25MHz	0
WNS	2.700ns	1.285ns	-1.415ns

Often, the algorithm represented with fixed-point arithmetic is used as part of an application. In this example, the buck converter model is used within an MCU-based power controller application where the control algorithm is implemented in software.

To enable nearly real-time development and debugging of the control algorithm, the complete system is simulated on FPGA. To attain full visibility to the system's IOs and state variables of algorithmic buck converter model, in a human-readable form, the IFA can be used. In the given example, the variables v_ in, v_ out, v_ snub, i_ ind and i_ snub were traced via IFA. The impact on resource utilization and timing with or without using the IFA and recording 2048 samples for each signal is depicted in table 1. While timing is only reduced by 2.23%, resource utilization of the trace instrumentation is increased by 87%. This is mainly due to additional bits required per variable and can be reduced by adapting the signal bit widths representing the fractional, integral, and exponential fragments of a variable.

V. Conclusion

We propose a method that streamlines the development of algorithms and designs involving the fixed-point format, while also facilitating the verification of fixed-point versus floating-point accuracy during simulation. Our approach to fixed-point variable declaration provides flexibility in detecting issues related to range and precision loss during simulation. It further offers configurable options for the fixed-point buffer width, catering to various FPGA platforms and easing computation-intensive operations. In addition, the Integrated Fixed-Point Analyzer (IFA) serves as a plug-and-play, cost-effective solution for debugging the fixed-point format during emulation. Notably, it operates independently of the fixed-point variable buffer and algorithm design. IFA enables real-time analysis of fixed-point values, empowering system behavior manipulation and the injection of faults. The case study presented in this paper serves as a testament to the effectiveness and efficiency of our proposed method.

Acknowledgment

The authors would like to thank the Federal Ministry of Education and Research (Germany) via the KI4BoardNet program for supporting this work.

References

[1] S. Herbst (2021) SVREAL-Synthesizable real number library in SystemVerilog, supporting both fixed and floating-point formats. [Online]. Available: https://git.io/svreal.
[2] S. Herbst et al., "An Open-Source Framework for FPGA Emulation of Analog/Mixed-Signal Integrated Circuit Designs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (Volume: 41, Issue: 7, July 2022), August 2021.
[3] ILA-Integrated Logic Analyzer [Online]. Available: https://www.xilinx.com/products/intellectual-property/ila.html.
[4] VIO-Virtual Input/Output [Online]. Available: https://www.xilinx.com/products/intellectual-property/vio.html.
[5] Don Lahiru "Integer vs. Floating-Point Processing on Modern FPGA Technology", Computing and Communication Workshop and Conference (CCWC), March 2020.
[6] Vivado Synthesis User Guide. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/sw manuals/xilinx2021 2/ug901-vivadosynthesis.pdf
[7] Verilog Fixed Point Arithmetic Modules. [Online]. Available: https://opencores.org/projects/fixed point arithmetic parameterized
[8] Support for ieeef?ixed pkg in Vivado for VHDL 2008 [Online]. Available: https://support.xilinx.com/s/question/0D52E00006hpbL3SAI/cannot-findfixedpkginieeeorieeeproposed
[9] Fixed point libraries support in VHDL. [Online]. Available: https://github.com/FPHDL/fphdl
[10] MathWorks's Fixed-Point Designer. [Online]. Available: https://in.mathworks.com/products/fixed-point-designer.html
[11] Vitis HLS's ap fixed type. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Fixed-Point-Data-Types
[12] S. Herbst (2021) msdsl. [Online]. Available: https://git.io/msdsl
[13] Wei-Chau Xie (2010) "Differential Equations for Engineers," Cambridge University Press (Chapter 8), June 2010
[14] G. Rutsch et al., "Boosting mixed-signal design productivity with FPGA-based methods throughout the chip design process," Design and Verification Conference in Europe, 2020.