End-to-End Framework for Novel Datatype Arithmetic Verification

Qiuwen Lou, Bing Ji, Stevo Bailey, Nilabja Chattopadhyay, Deepak Shivarudrappa and Sankalp Dayal
Amazon.com LLC, 1100 Enterprise Way, Sunnyvale, CA, United States of America
{loqiuwen, jibing, stebi, nchattop, shivaru, sankalpd}@amazon.com

***Abstract*—Machine Learning (ML) accelerators are increasingly adopting diverse datatypes and data formats, such as FP16 and microscaling, to optimize key performance metrics such as inference accuracy, latency and power consumption. However, hardware modules like the arithmetic units and signal processing blocks associated with these datatypes pose unique verification challenges. In this work, we present an end-to-end flow for this novel datatype verification. In RTL level, we delve into Datapath Validation[1] (DPV), a formal verification approach that has proven instrumental in tackling the verification challenges of the datapath verification with new datatypes. We use DPV along with simulation-based approach for our datapath verification. We use FP16 as a case study here to discuss our approaches, but also demonstrated how our methodology can be extended to more complex data types such as microscaling. In application level, we leverage PyTorch-C based approach to verify the accuracy in the context of the neural network. Additionally, we further extend the tool for rapid performance projection for various operators. By leveraging the mathematical precision and rigor of formal verification as well as our customized C Reference model implementation, we were able to identify and address critical issues with lightning-fast speed, leading to a more reliable final product. As an example, we verified the FP16 implementation in under 8 weeks—a remarkable feat that would have been a daunting task for traditional simulation-based methods. The final application-level accuracy difference is within 0.1% of the original PyTorch model. We proved that the power of the DPV can lead to faster time-to-market and increased confidence in the IP's integration.**

I. Introduction

Many machine learning (ML) models, especially large language models, are notorious for their vast number of parameters and computations [2]. To decrease both, researchers have been actively working on reducing precision while maintaining model accuracy [3]. One such method uses new floating-point data types, shrinking the sizes of the mantissa and exponent, while keeping the benefit of a much wider range over integer data types. Most CPUs and many ML accelerators support FP32 already, but upcasting these new data types to FP32 loses the performance benefits of reduced precision. Hence, there is a need for custom hardware to support these new data types.

Direct hardware support for new data types requires changing both compute logic to leverage reduced precision and control logic to maximize the performance of simpler operations. Creating new hardware to keep up with cutting-edge research suggests adopting an agile design methodology, but this is challenging in practice. The biggest cost stems from a long time-to-market, as any hardware change needs verification, physical design, tape-out testing, and validation. Numerous verification and validation efforts are required to test all combinations of operations. There are also lots of corner cases. Therefore, optimizing the verification flow will lead to faster time-to-market.

In this paper, we propose a novel verification framework. Key features of the framework include the following.

1) A C reference model verifies the RTL behavior. In both the reference model and RTL, the design is modular and easily adapts to any new data type.
2) Custom interfaces of the RTL and C reference model achieve data correctness checks with a single build.
3) PyTorch is bound to the C reference model, supporting early application-level accuracy checks and facilitating HW/SW/Science co-design.
4) The formal verification tool (Synopsys DPV [1]) is reused to measure function-specific performance, eliminating the need for additional time-consuming simulations.

These features show the framework's benefit to both verification and ML software development. We applied the framework to our own accelerator, and it shortened our hardware delivery time and guaranteed the quality of the new hardware. Specifically, the DPV tool identified 11 bugs across the RTL and C model. We employed various strategies, such as case splitting and hierarchical proof, to achieve convergence for all planned FP16 functions in the DPV tool. Some bugs, particularly those in corner cases, would have been challenging to detect through random simulation alone. At the ML application level, we use LLAMA and Vision Transformer (ViT) networks to evaluate the accuracy of the data path. Our hardware achieves excellent accuracy, coming within 0.1% of PyTorch's end-to-end performance. This framework has significantly reduced our development time from six months to two months compared to the traditional approach. We are planning to extend this framework to facilitate the design of other proposed data types, such as FP4, FP6, FP8, and advanced data formats like microscaling.

II. Background

A. Data Types

Recent advancements in machine learning hardware have introduced several new floating-point (FP) and integer (INT) data types tailored to optimize model efficiency and reduce memory usage. These data types include FP4 and FP6, which use fewer bits than traditional FP32, reducing precision slightly but offering substantial benefits in speed and power efficiency. FP8, another emerging data type, strikes a balance between efficiency and accuracy, allowing for faster computations without a significant loss in model performance. Floating point types, defined by the IEEE-754 standard, store a significand and mantissa separately, which greatly increases the range while reducing the precision and creating a nonlinear distribution of values. The adoption of these new data types allows machine learning models to scale efficiently, making it feasible to deploy large models in constrained environments.

B. Datapath Validation

Datapath Validation (DPV) is a formal verification methodology that mathematically proves the functional equivalence between a high-level reference model (such as a C++ implementation) and its corresponding hardware design. DPV focuses on verifying the datapath elements responsible for processing and manipulating data, in particular complex arithmetic operations such as multiply and square root. By exhaustively exploring all possible input combinations within the model's scope, DPV ensures that the hardware design performs exactly as intended, adhering strictly to standards like IEEE-754. This approach provides exhaustive coverage and early detection of errors but may face scalability challenges and requires specialized expertise in formal methods.

III. Problem Statement

In this work, we use FP16 (16-bit floating point) as a case study. Verification of FP16 implementations presents significant challenges due to the complexity of floating-point arithmetic and the stringent correctness requirements dictated by the IEEE-754 standard. Ensuring that a hardware Register Transfer Level (RTL) implementation accurately reflects its high-level C++ reference model is critical for system reliability, especially in applications where precision and correctness are paramount.

We evaluated two options to support the standard FP16 data type, the standard GCC built-in library and open-source libraries. In this work, we selected the SoftFloat open-source library [4] for the following reasons.

1) *C++ Standard Compatibility:* The C++ standard only introduces support for the fp16 data type in the C++23 version. However, the DPV tool is currently limited to gcc-13.2.0 and gcc-7.3.0, which is not C++23 compatible.
2) *Enhanced Tool Compatibility:* Synopsys has optimized and provided formal tool-friendly versions of the SoftFloat libraries. These optimizations significantly improve DPV convergence, making SoftFloat a more suitable choice for our requirements.

A. Complexity of FP16 Verification

We leveraged the standard operator suite from IEEE-754 for our design. There are in total 13 functions. We categorized them as following. We also support various rounding mode per SoftFloat spec.

1) Standard compute functions. This includes ADD, MUL, DIV, SQRT, FMA (fused multiply and add), MULS (FP16 $\times$ FP32).
2) Non-compute functions. This include comparison, sign, MIN/MAX.
3) Integer to floating point conversion functions. That include various integer format to/from floating point format.

FP16 arithmetic operations involve operands that are 16 bits wide, leading to an enormous input space. For binary operations such as addition or multiplication, the total number of possible input combinations is $2^{16} \times 2^{16}$, or 4,294,967,296. Exhaustively testing all these combinations through simulation is impractical due to time and resource constraints. Several factors contribute to the verification complexity.

- *Precision and Rounding Challenges:* IEEE-754 standard has some flexibility in implementation. Different implementations may support different handling of rounding modes, exception flags, and denormalized numbers. Verifying that the implementation correctly handles these aspects is non-trivial.
- *Special Case Handling:* The FP16 format includes special values such as NaNs (Not a Number), positive and negative infinities, zeros, and subnormal numbers. Each of these requires careful verification to ensure correct behavior in all scenarios.
- *IEEE-754 Compliance:* Strict adherence to the IEEE-754 standard is mandatory for floating-point implementations. We ensure the output matches the PyTorch outputs.

B. Pros and Cons of Verification Methodologies

A summary of the pros and cons of both constrained random simulations and formal verification methodologies guides the development of the combined verification strategy and can be found in Table I.

IV. Framework

In this section, we summarize our framework and proposed design. Our framework has two parts, as seen in Figure 1. One is the FP16 IP-level verification, and the other is the application-level verification.

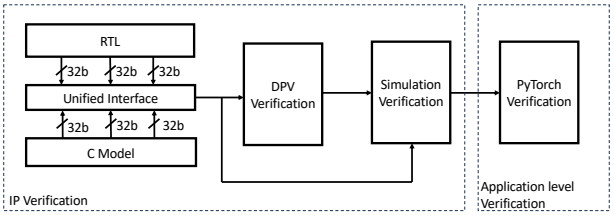

Fig. 1. High-Level Framework Diagram

A. FP16 IP-Level Verification

1) Modularized RTL and C Reference model for FP16 Verification: The FP16 module in this work is used for elementwise operations. In the architecture and RTL design phase, we (1) created a new module for all the FP16-related designs, and we also (2) generated submodules within the module for both the RTL and C reference models for FP16. The design is highly scalable. This also helps lay the groundwork to leverage both the DPV and simulation in our later verification efforts as it is easier for DPV to identify the blocks and converge.

Our previous accelerator only supported integer data formats, so we added a separate hardware unit to support FP16 data formats. Both integer and floating point units can run in parallel. We also define the interface between the integer and FP16 units, and we include this interface in the verification device under test. As such, in the verification efforts, we only need to verify the FP16-related instructions and their interfaces with the existing integer datapath.

The floating point hardware block includes four processing elements, namely ADD/MUL, DIV/SQRT, non-computational, and conversion, as seen in Figure 2. The ADD/MUL unit handles additions, multiplications, and fused-multiply-add (FMA). It supports FP32 multiplication for the MULS instruction, and a separate converter upcasts one input from FP16 to FP32 for this instruction. The DIV/SQRT unit handles both divide and square root. The non-computational unit handles sign manipulations and comparisons. The conversion unit handles conversions between FP16 and integer formats. In our accelerator, each instruction typically uses one blocks in the figure. So, to reduce area, all processing elements reuse hardware resources when possible, such as the divider and square root sharing an iterative datapath.

Our C Reference model contains two sets of functions, the functions for the hardware modules, and the functions for executing instructions. We map the module to C functions directly to ensure a tight coupling between the reference model and RTL. As such, for each of the blocks described above, we implemented a C function, which

TABLE I
PROS AND CONS OF VERIFICATION METHODOLOGIES

Verification Methodology	Pros	Cons
Formal Application DPV	1) *Exhaustive Coverage:* Provides mathematical proof of correctness for all input combinations within the scope of the model. 2) *Early Detection of Design Flaws:* Identifies issues early in the development cycle, reducing downstream debugging efforts.	1) *Scalability Issues:* Computational demands increase exponentially with design complexity, potentially limiting applicability. 2) *Specialized Skill Set Required:* Requires expertise in formal methods and tools, which may necessitate additional training. 3) *Tool Limitations:* Vendor Tools may have limtations supporting libraries typically used in HLM reference models.
Constrained Random Simulations	1) *Realistic Testing Environment:* Simulates the design under a variety of operating conditions, revealing issues that occur in practical scenarios. 2) *Accessibility:* Easier to set up and execute without requiring deep formal verification knowledge.	1) *Incomplete Coverage:* Cannot guarantee that all possible input combinations or rare corner cases are exercised. 2) *Resource Intensive:* Large numbers of simulations consume significant computational resources and time.

has the same module as RTL. The details can be seen in Figure 3. This helps us to converge when using DPV for the design. However, we also want the C reference model to simulate the hardware behavior, so we implement functions to simulate the instructions in our accelerators. These functions call the underlying modules.

By leveraging the above approach, DPV can converge promptly. Our efforts also allow the C reference model to achieve bit-level accuracy with the RTL.

2) Customized Interface for RTL and C Reference Model Compilation: We unify the interfaces of the RTL and C models to make the design scalable. We match both the number of inputs and the bitwidths. To make our framework general to support any novel, low precision data type, we upcast the data type of interest to 32 bits in the C reference model interface. In RTL, we make a wrapper on top of the port to support up to 32-bit inputs. We achieve this by time multiplexing. Our RTL dynamically interprets these inputs, extracting and utilizing the relevant bits based on the specified function selection. This approach ensures efficient and accurate processing of diverse computational tasks while maintaining a streamlined and flexible API design.

We also unified the number of inputs and outputs in the interface to make sure that C reference model matches the RTL. In this design, we support a maximum of three inputs and one output. In general use cases, this approach allows us to accept any data input configuration required by the different FP16 operations. The interface includes control inputs, data inputs, and outputs for various operations.

B. Application-Level Accuracy Verification

We also established the impact of hardware changes and optimizations on machine learning application or model-level accuracy. This is done by running inference on complete deep learning models and evaluating their outputs.

After IP-level verification, as described in the previous section, we leverage the C reference model to execute the same operators in the inference flow. The inference flow is run using PyTorch. The C reference operators are wrapped with Python bindings for integration in PyTorch. We use the standard python package ctypes for this. There are other options, such as PyBind11 and FFI (Foreign Function Interface), which achieve the same purpose, but we select ctypes because it is the simplest method. The bindings extract pointers to binary buffers of numpy objects and route the function calls to the C reference implementation. The C reference is compiled into a shared object, which is loaded in the python environment at runtime. Control is also provided to the developers to be able delegate only a subset of operators of interest. This integration has enabled HW/SW/Science co-design, where we are able to gain confidence on the applicability of the HW compute datapath and the impact of this on the user-facing application accuracy KPIs.

C. Performance Estimation with DPV

While DPV tools traditionally focus on verifying data correctness by comparing RTL output with reference models, we found DPV can also be used to encompass RTL performance testing. In our FP16 RTL implementation, we utilize a ready-valid protocol for output handshaking. Our novel approach instructs the tool to pull down the “ready" input signal, simulating back pressure on the RTL and preventing continuous output. The tool then samples the “valid" signal at intervals corresponding to the desired latency duration specified in the microarchitecture specification. If the “valid" signal fails to assert within the designed number of cycles, it indicates that the RTL cannot meet the stipulated latency requirement.

This innovative method offers several significant advantages. (1) It provides function-specific latency verification, flagging issues for each distinct function code. (2) It demonstrates high sensitivity to design changes, ensuring that performance impacts are quickly identified. (3) It eliminates the need for additional time-consuming performance tests in simulation, streamlining the verification process.

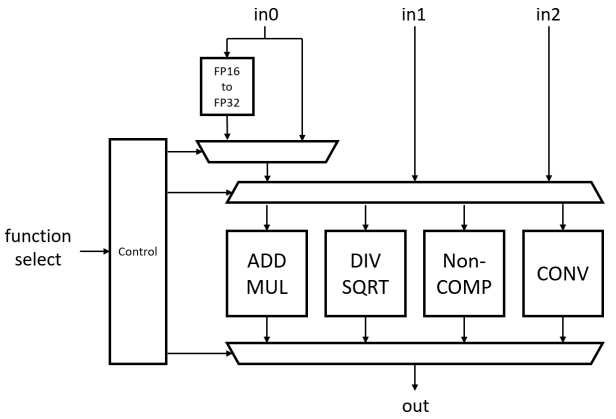

Fig. 2. RTL Block Diagram

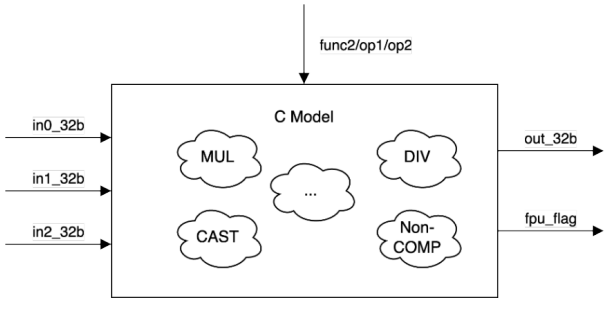

Fig. 3. C Reference Model Diagram

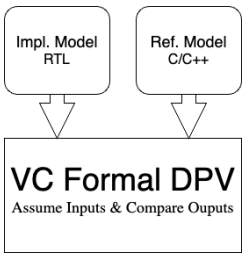

Fig. 4. DPV Diagram

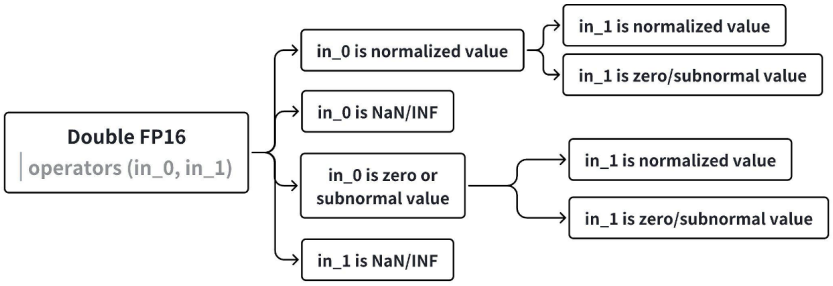

Fig. 5. CaseSplit Diagram

The following code snippet illustrates how we implement this approach:

```
if {$func_type == "ADD" || $func_type == "MUL" || $func_type == "FMA" ||
    $func_type == "FNMA" || $func_type == "MULS"} {
    lemma valid_out = impl.valid_out(9) == 1
} elseif {$func_type == "DIV" || $func_type == "SQRT"} {
    lemma valid_out = impl.valid_out(25) == 1
} else {
    lemma valid_out = impl.valid_out(7) == 1
}
```

V. Deploying DPV

A. Framework Overview

We designed a flexible and efficient setup that addresses the challenges of verifying multiple FP16 functions using the datapath validation (DPV) formal tool. The key components of our framework include

1) a versatile RTL interface that multiplexes different input combinations based on the function value before feeding them into the main FP16 RTL compute logic,
2) a unified C model with three 32-bit wide inputs that are parsed differently based on the function value, ensuring consistency across simulation and formal verification, and
3) a top-level tool command language (TCL) file that dynamically maps inputs between the RTL interface and C model, customized for each function.

This innovative setup achieves two primary goals.

1) A single RTL build can execute different functions by simply adjusting the function value in the top TCL file and RTL interface, eliminating the need for multiple compilations.
2) The same C model is seamlessly reused in both simulation and DPV environments, ensuring consistency and reducing development effort.

By leveraging this approach, we significantly reduce compilation time and enable parallel execution of all functions on the server. This not only accelerates the verification process but also allows for more comprehensive coverage of the FP16 module's functionality. The flexibility of our framework facilitates easy addition of new functions or modifications to existing ones, making it a scalable solution for ongoing development and verification efforts.

1) RTL Interface: We created an interface to multiplex different input combinations based on the function value before feeding them into the main FP16 RTL compute logic. The interface is designed to support a maximum of three inputs with various data types including (U)INT8, (U)INT16, (U)INT32, FP16, and FP32. To accommodate this flexibility, we use six 16-bit input signals (data1_in1, data1_in2, data1_in3, data2_in1, data2_in2, data2_in3) which can be combined as needed based on the function selection. This approach allows us to accept any data input configuration required by the different FP16 operations. The interface includes control inputs, data inputs, and outputs for various operations. A snippet of the SystemVerilog RTL is shown below:

```
module fpu (
  // function select inputs
  input  logic [3:0]  func2,
  input  logic [7:0]  op1,
  input  logic [7:0]  op2,
  // data inputs
  input  logic [15:0] data1_in1,
  input  logic [15:0] data1_in2,
  input  logic [15:0] data2_in1,
  input  logic [15:0] data2_in2,
  // ... [additional inputs and outputs]
);
```

This flexible input structure allows us to handle various data types and input configurations within a single interface, simplifying the verification process across different FP16 functions.

2) C Model API: All input parameters (in0_32b, in1_32b, in2_32b) are uniformly 32 bits wide, providing a consistent interface across various operations. The model's internal logic dynamically interprets these inputs, extracting and utilizing the relevant bits based on the specified function selection. This approach ensures efficient and accurate processing of diverse computational tasks while maintaining a streamlined and flexible API design.

```
bool ewop_fpu(int vl, int func2, int op1, int op2,
              int32_t in0_32b, int32_t in1_32b, int32_t in2_32b,
              int32_t &out_32b, uint8_t &fpu_flag);
```

3) Top-Level TCL: The top-level TCL file dynamically establishes mappings between the RTL interface and C model for various functions. For instance, consider the MULS function (FP32 * FP16). The TCL file creates a connection that combines two 16-bit distinct inputs from the RTL interface (data1_in1 and data1_in2) and maps them into a single, consolidated 32-bit input in the C model, to accommodate the FP32 input.

```
assume asum1 = -always (impl.data1_in1[15:0] == spec.in0_32b[15:0])
assume asum2 = -always (impl.data1_in2[15:0] == spec.in1_32b[15:0])
assume asum5 = -always (impl.data2_in1[15:0] == spec.in0_32b[31:16])
```

B. Facilitating Convergence

We needed to verify 13 complex functions as discussed in Sec.IIIA, including operations like square root and multiplication. Without any convergence strategy, most functions either ran for hours or failed to converge, which was due to the massive data spaces or complex design pipelines. This section demonstrates our innovative approach to achieving full convergence using the DPV tool, focusing on the multiplication function as an example.

1) Case Splitting Strategy: The case splitting strategy involves decomposing complex properties or assertions into smaller, more manageable sub-cases. For the FP16 multiply function, we categorized the multiplicand and multiplier into six distinct combinations based on their nature (normalized, zero/subnormal, NaN, or infinity). By applying this strategy, we directed the DPV tool to focus on each case individually. This approach allowed the tool to run sub-cases in parallel, optimizing computational and memory resources and resulting in faster convergence. Furthermore, if a sub-case failed, it became significantly easier to pinpoint the root cause. The following TCL code illustrates the case splitting approach for normal and subnormal cases.

```
proc case_split_double_operators {} {
    caseSplitStrategy basic

    caseBegin dnorm_norm_16
    # in_0 is zero or subnormal value
    caseAssume (spec.in0_32b(1)[14:10] == 5'h00)
    # in_1 is normalized value
    caseAssume (spec.in1_32b(1)[14:10] != 5'h00)
    caseAssume (spec.in1_32b(1)[14:10] != 5'h1f)
    ...
    caseBegin norm_norm_16
    # in_0 is normalized value
    caseAssume (spec.in0_32b(1)[14:10] != 5'h00)
    caseAssume (spec.in0_32b(1)[14:10] != 5'h1f)
    # in_1 is normalized value
    caseAssume (spec.in1_32b(1)[14:10] != 5'h00)
    caseAssume (spec.in1_32b(1)[14:10] != 5'h1f)
}
```

2) Assume-Guarantee (Hierarchical Proofs) Strategy: We implemented an assume-guarantee strategy to break down complex tasks into a hierarchy of smaller subproblems. For the MUL/FMA functions, we first focused on verifying by cutting off the multiply operator deep inside the RTL submodule. We add assume and lemma checks to ensure the multiply operator produces the proper output value based on the multiplicand and multiplier inputs.

```
set submodule_path "impl.u_fpunew.gen_operation_groups[0].i_opgroup_block.\
                    gen_merged_slice.i_multifmt_slice.gen_num_lanes[0].\
                    active_lane.lane_instance.i_fpnew_fma_multi"

cutpoint mpier = ${submodule_path}.mantissa_a(6)
cutpoint mpcand = ${submodule_path}.mantissa_b(6)

lemma check_mul = ${submodule_path}.product(6) == mpier * mpcand
```

TABLE II
BUGS EXPOSED BY DPV

Type of Bug	Notes	DPV Effort (in days)	Found in Simulation
Output data mismatch in FP16 × FP32 MULS function	• Input data is FP32(3803609600) and FP16 (-0.00001723). • The output data from the C model is 0xfc00 (-Inf), but the RTL output is 0xfbff (-65504). • Using FP32 as an intermediate data type cannot provide sufficient data accuracy.	3	N
FPU exception flag (Invalid/Inexact) mismatch in F2INT8 Function	• Flag mismatch in the F2INT8 function with input data -128.5 (0xd804). The RTL raises an invalid flag, while the C model reports an inexact flag.	0.5	N
FPU exception flag (Invalid) mismatch in FP16 × FP32 MULS function	• Invalid flag mismatch for input data 0x0000_1217 and 0x7d6b. • C model set the invalid flag because one of input data is a qNaN, but RTL did not.	0.5	N
FPU exception flag (Underflow) mismatch in DIV function	• Underflow flag mismatch for DIV (0.00012202, -2). • C model reported underflow and inexact flags, but RTL only reports inexact flag.	3	N
Output data mismatch in DIV function	• Mismatch in DIV function with input data 0x350 and 0xd000.	0.5	N
Output data mismatch in SQRT function	• Output data mismatch with input data is 0x1. • C model generates reference as 0xc00, but the RTL outputs 0x7e00.	2	N
FPU exception flag (Underflow) mismatch in FP16 × FP16 MUL function	• RTL implementation incorrectly set the underflow flag in a specific scenario with input data 0x3fa5_d66a and 0x2ef. • Both the RTL and the C model have the same output value, which is within the FP16 range, but the RTL unexpectedly sets the underflow flag.	3	N

After achieving convergence for the multiply operation from the submodule, we add top-level assumptions to guide the tool in treating multiply as a guaranteed output. Thus, verifying the remaining steps of normalization and rounding, and addition for FMA, is easier.

```
assume ${submodule_path}.product(6) == \
       ${submodule_path}.mantissa_a(6) * \
       ${submodule_path}.mantissa_b(6)
```

By combining case splitting and assume-guarantee strategies, we successfully achieved convergence in the multiply and FMA functions. This approach enabled us to fully verify output correctness across all input combinations efficiently, demonstrating the effectiveness of our methodology in handling complex floating-point operations.

VI. RESULTS

```
   localparam int          signed   INT_MAX = (1 << (INT_WIDTH-1))-1;
   localparam int        unsigned  UINT_MAX = 1 << INT_WIDTH;
-- localparam int          signed   INT_MIN = -1*(1 << (INT_WIDTH-1));
++ localparam longint unsigned   INT_MIN = (1 << WIDTH)-(1 << (INT_WIDTH-1));

   if (IntFmtConfig[ifmt]) begin : active_format
     always_comb begin : detect_overflow
       of_after_round = 1'b0;
       // negative overflow (unsigned)
--     if (!signed & (rounded_int < 0))
--       of_after_round = 1'b1;
--     // negative overflow (signed)
--     else if (signed & (rounded_int < INT_MIN))
++     if (!signed & rounded_sign & !(rounded_int == 0))
         of_after_round = 1'b1;
       // positive overflow (unsigned)
--     else if (!signed & (rounded_int > INT_MAX))
++     else if (!signed & !rounded_sign & (rounded_int >= UINT_MAX))
++       of_after_round = 1'b1;
++     // negative overflow (signed)
++     else if (signed & rounded_sign & (rounded_int < INT_MIN) & !(rounded_int == 0))
         of_after_round = 1'b1;
       // positive overflow (signed)
--     else if (signed & !(rounded_int > UINT_MAX))
++     else if (signed & !rounded_sign & (rounded_int > INT_MAX))
         of_after_round = 1'b1;
       end
     end
```

A summary of bugs exposed by DPV is presented in Table II. The team began parallel simulation and formal verification efforts after completing the C model implementation. We spent five weeks configuring the formal verification (DPV) tool and exploring convergence strategies for different functions. During this DPV setup period, simulation testing did not reveal those bugs. While constraining input data randomization might have helped hitting issues sooner, simulation alone could not provide the same level of confidence as formal verification for catching and fixing bugs. The code snippet below it shows an example of fixing an overflow flag bug when casting to an integer. We were able to generate 100% convergence between our reference model implementation and RTL for each supported operation in 8 weeks.

- Test Plan and C-Model Development: 3 weeks (portable to simulation based verification environment)
- DPV Tool Setup: 1 week
- Runtime and convergence strategies: 4 weeks.

After integrating with PyTorch, we evaluated the design on LLAMA and ViT. For LLAMA, we use perplexity score. For ViT, we leverage the classification results for ImageNet. We found that both the perplexity score and classification results are within 0.1% compared with running PyTorch end-to-end. That shows the datapath we developed is very close to PyTorch — our golden reference. The framework also significantly reduced our development time. Two months after the RTL freeze, we already have confidence that our RTL can support our models of interest. The process typically takes longer than 6 months without our framework.

VII. Conclusion

In this work, we demonstrate our new verification framework for new data types. Four techniques, including Modularized RTL and C Reference Model for FP16 Verification, Customized Interface for RTL and C Reference Model Compilation, Integration with PyTorch for Application Level Accuracy Verification and Performance Estimation with DPV are proposed within the framework. The framework can be used to significantly reduce the development cycle when there is a new, low precision data type. We also plan to use DPV to generate RTL coverage which we could in turn merge with coverage generated from simulation-based test benches.

References

[1] M. Bernard *et al.*, *Finding Your way Through Formal Verification*, 2nd ed. SemiWiki LLC Danville, CA, 2023.

[2] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," in *arXiv preprint arXiv:2302.13971*, 2023.

[3] G. Xiao *et al.*, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *International Conference on Machine Learning. PMLR*, 2023.

[4] J. Hauser *et al.*, "Softfloat," in *https://www.jhauser.us/arithmetic/SoftFloat.html*, 2024.