TrueFloat: a templatized arithmetic library for HLS floating-point operators*

Michele Fiorito, Serena Curzel, Fabrizio Ferrandi

Politecnico di Milano, Piazza Leonardo da Vinci 32 20133 Milano, Italy
michele.fiorito@polimi.it, serena.curzel@polimi.it,
fabrizio.ferrandi@polimi.it

Abstract. Hardware designers working on FPGA accelerators are free to explore ad-hoc value representations that differ from the IEEE 754 floating-point standard, significantly reducing resource utilization and latency. In fact, while some applications are amenable to fixed-point quantization, others may require a wider dynamic range of values, better represented through a customized floating-point encoding. TrueFloat automates the process of designing accelerators with custom floating-point representations by introducing a methodology for the generation of customized floating-point units within a state-of-the-art High-Level Synthesis tool, providing high performance and fast prototyping. With TrueFloat, it is possible to translate a software description with standard floating-point calculations into an optimized hardware design featuring any number of different value encodings. Generated floating-point units are competitive with respect to state-of-the-art templatized libraries.

Keywords: High-Level Synthesis · FPGA · Custom data types.

1 Introduction

When a software description is translated into a hardware accelerator for Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs), developers have the opportunity to customize data types to balance latency, power consumption, and computational precision. If the accelerator is designed in Verilog/VHDL, floating-point calculations can be manually transformed into fixed-point (quantization) or into floating-point formats tailored to the specific application. With High-Level Synthesis (HLS), developers have the opportunity to describe the accelerators at a higher level of abstraction (C/C++ code) and increase their productivity; however, experiments with custom data types are limited by the back-end libraries supported by the HLS tool, which are mostly focused on fixed-point types.

In this paper, we present TrueFloat: an extensible framework for the exploration of custom floating-point data types and the automated synthesis of

* This research was partially supported by the HERMES project - European Union's Horizon 2020 research and innovation programme, grant agreement N. 101004203.

corresponding optimized arithmetic units. TrueFloat is embedded into the open-source HLS tool Bambu [3], and it provides effortless translation between different floating-point encodings through simple command-line options, integration with other optimization techniques present in the HLS flow, and the possibility of generating multi-precision accelerators where different floating-point encodings are used in different parts of the design.

2 State of the art

VFLOAT [7] is a VHDL library containing variable-precision floating-point cores for basic arithmetic operations, along with conversion operators to and from fixed-point representations; the components are parametrized, but they cannot be adapted to different frequency domains or different target FPGAs. The FloPoCo framework [2] provides a wide range of features to generate, optimize, and test complex arithmetic functions for low-level accelerator design; it can generate templatized VHDL cores for integer, fixed-point, floating-point, and complex types, which can be further optimized for a specific FPGA and frequency target. When dealing with floating-point data types, FloPoCo uses a proprietary encoding, different from the IEEE standard, and optimized to allow easier exception handling. Both approaches require the user to integrate each RTL core in the accelerator design manually.

If the customization of floating-point computation is available at a higher level of abstraction, the user can write a software description and feed it to an HLS tool. Proprietary libraries exist offering a C++ API to replace standard numerical types in the high-level software description before it is passed to their HLS tools, mapping C++ types onto a back-end RTL library during the synthesis process. This is the case of the Mentor Graphics Algorithmic C Datatypes [5], providing arbitrary precision integer, fixed-point, and floating-point types. A similar library, Template HLS [6], provides a unique implementation used for both simulation and synthesis.

The main strength of TrueFloat is the integration with an HLS tool, which unlocks new optimization opportunities and the possibility of generating an equivalent representation at a higher level of abstraction. Existing approaches based on libraries of functional units cannot access such opportunities, and they require significant code rewriting, while TrueFloat users can write their applications with standard floating-point operations and the tool will automatically generate the requested custom units.

3 Proposed approach

TrueFloat integrates the customization of floating-point representations into the HLS flow of Bambu; it is not required to use custom data types in the input code, as standard floating-point operations and types will be replaced with custom representations during the HLS flow, according to user-defined constraints expressed

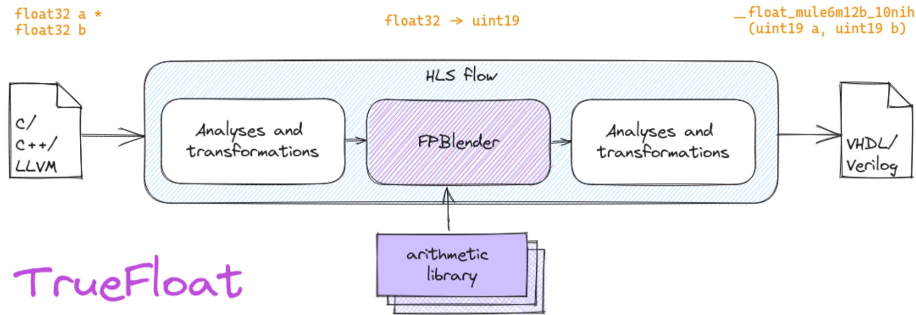


Fig. 1: TrueFloat synthesis methodology.

as command-line options. Offloading the replacement from the application description phase to the HLS flow turns an error-prone, complex procedure into a reliable process that autonomously handles types replacement, conversions, and custom arithmetic units generation. We introduced floating-point manipulation as a new compiler step within the HLS flow, called FPBlender, so that it can exploit information generated during previous analysis steps on the intermediate representation of the input program and apply more accurate optimizations after the custom floating-point implementation has been applied (Figure 1).

TrueFloat features a fully templatized IEEE 754-like floating-point representation, which allows the definition of any bitwidth for both mantissa and exponent, provided that together with an optional sign bit the new data type can fit into a 64-bit word. It is also possible to arbitrarily define the exponent bias, allowing accurate centering of the representation over the range of values that need to be encoded; fixed-point values can thus also be encoded by forcing a zero-bit exponent and using the bias value to set the decimal point. It is possible to encode both standard and subnormal values, but also standard-only or subnormal-only values. Exception encoding is customizable, allowing to choose between standard IEEE 754 exceptions and a representation where floating-point operators will simply avoid exception handling; such a configuration is useful when the application is not expected to fall into exceptional behaviors at runtime. Finally, it is possible to switch from round to nearest even to a simpler truncation rounding mode when small errors are acceptable.

The floating-point lowering pass FPBlender translates each floating-point variable or operation present in the input code into its customized counterpart operating on the intermediate representation of the HLS tool. FPBlender generates ad-hoc functional units exploiting the TrueFloat library of templatized components, which currently contains implementations for basic arithmetic operators (addition, subtraction, multiplication, division, and comparison) and bidirectional type conversion operators (floating-point to integer, integer to floating-point, and floating-point to floating-point). The floating-point functions in the TrueFloat library are soft-float implementations built from basic integer operations; input and output parameters are defined as unsigned integers as well.

After the modifications are applied, the intermediate representation no longer contains floating-point types, and floating-point operations are replaced with calls to corresponding functions from the TrueFloat library.

One key feature of TrueFloat is the ability to exploit optimizations both during the front-end compilation phase and the back-end synthesis flow. Before the execution of FPBlender, the intermediate representation contains floating-point operations as single instructions, simplifying data flow analysis and value range analysis. After the lowering has been applied, several HLS passes can be exploited to optimize the intermediate representation; inter-procedural analysis to propagate function parameters and return values is particularly effective during this phase as floating-point operations have been replaced with function calls.

Finally, TrueFloat can generate a C equivalent of the HLS intermediate representation along with the final RTL design. The C output is a one-to-one equivalent of the intermediate representation converted into RTL code, and thus it can be exploited to perform fast and accurate software simulations of the customized floating-point operations.

4 Implementation

4.1 FPBlender step

FPBlender generates custom floating-point arithmetic operators and integrates them into the Bambu intermediate representation, adding all necessary interfaces with the rest of the design; it has been implemented as a function scope transformation so that different data types can be specified for each function in the input application. The first phase in the FPBlender lowering process analyzes the call graph of the application, propagating information about custom floating-point types and understanding where type conversions are required. Subsequently, each function body is analyzed and annotated to identify variables that will be converted into custom types, while floating-point instructions are replaced with calls to templatized functions from the library. The actual arithmetic cores are generated through a versioning process where the function from the library receives a set of specialization arguments and a new name formed appending the specialization string to its base name. Finally, a second function-scope transformation is applied where annotated variables are converted to unsigned integer types with the exact bitwidth needed to contain the custom floating-point encoding, and conversion operators are inserted where necessary. At this point, the lowering is completed: all floating-point types and arithmetic operations have been replaced by bit manipulation and integer arithmetic instructions, and the subsequent HLS analysis and optimizations passes are executed on the updated intermediate representation.

4.2 TrueFloat arithmetic library

The TrueFloat library is composed of three types of operators (arithmetic operators, comparators, and converters) implemented in C following a custom IEEE

```
1 float myAdd(float a, float b)
2 {
3   unsigned long long _a = __float_to_e6m12b_63noh(*((
        unsigned int*)&a), spec);
4   unsigned long long _b = __float_to_e6m12b_63noh(*((
        unsigned int*)&b), spec);
5   unsigned long long _res = __adde6m12b_63noh(_a, _b, spec);
6   unsigned int _out = __to_e8m23b_127nih(_res, spec);
7   reutrn *((float*)&_out);
8 }
```

Fig. 2: C equivalent of the intermediate representation after FPBlender.

754 encoding class and pre-compiled inside the Bambu distribution package. All functions have arguments representing standard operands followed by a set of eight specialization arguments to indicate the number of exponent bits, fractional bits, the exponent bias, the rounding mode, the exception mode, whether hidden one is enabled, whether subnormals are enabled, and the sign mode.

4.3 HLS transformations

TrueFloat takes advantage of the Bambu HLS engine to perform constant propagation, function versioning, function inlining, and cyclic inter-procedural analysis. The core transformation which is necessary for the specialization of TrueFloat functions is constant propagation: after a library function is versioned by the FPBlender transformation step, it is always called with the same set of specialization arguments, so all sub-expressions involving specialization arguments can be resolved at compile-time and unnecessary instructions can be removed.

Bambu also features inter-procedural optimizations; the most effective ones for the TrueFloat flow are bit-value inference [1] and value range analysis [4]. Inter-procedural analysis may be able to detect that an operator is never fed with values encoding exceptions, like NaN or infinity, so Bambu may remove the related checks from the final operator resulting in a faster and smaller design.

Finally, the standard HLS steps of scheduling, resource allocation, and module binding are applied. Unlike approaches based on libraries of black-box components, TrueFloat generates functional units that go through the full HLS flow, granting Bambu an accurate timing model of each component and the possibility of applying retiming techniques to remove input/output registers.

4.4 Example

We will now show a simple example of the TrueFloat conversion flow assuming that the function to be synthesized returns the sum of two floating point values, and that a specialization string is applied to indicate that we require 6 bits for the exponent, 16 bits for the mantissa, a bias of -63, round to nearest even rounding mode, no exception handling, and hidden-one representation.

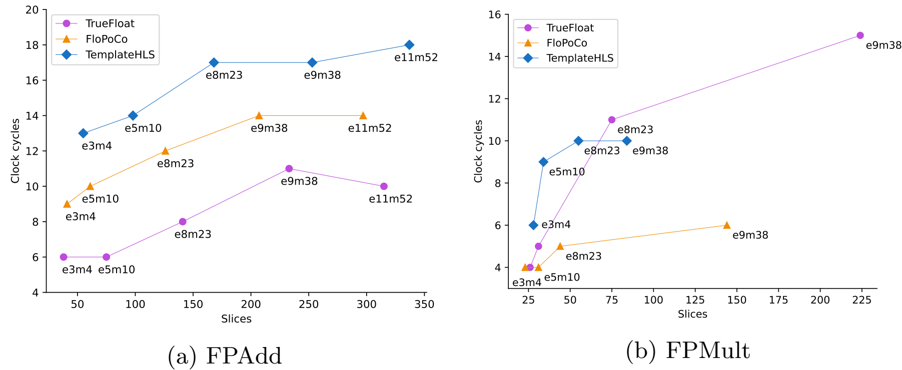
```
1 unsigned long long __float_adde6m12b_63noh(
2   unsigned long long a, unsigned long long b,
3   bits8 exp_bits, bits8 frac_bits,
4   int32 exp_bias, bits8 rnd, bits8 exc,
5   flag one, flag subnorm, bits8 sign)
6 {
7  ...
8  if(rnd == RND_NEVEN)
9  {
10    LSB_bit    = (RSig0 >> 3) & 1;
11    Guard_bit  = (RSig0 >> 2) & 1;
12    Round_bit  = (RSig0 >> 1) & 1;
13    Sticky_bit = (RSig0 & 1) | sb;
14    round  = Guard_bit & (LSB_bit | Round_bit | Sticky_bit);
15  }
16 ...
17  if(rnd)
18    Rrounded = RExp0RSig1 + round;
19  else
20    Rrounded = RExp0RSig1;
21  ...
```

Fig. 3: Partial implementation of the floating-point addition operator with unnecessary code eliminated after constant propagation.

After the FPBlender step, the intermediate representation looks as shown in Figure 2 (a C description is used for clarity, and specialization arguments have been collapsed into one): the standard floating-point addition has been replaced by its equivalent, versioned function call from the templatized library. Two conversion operators are added to convert the top function parameters from the standard floating-point encoding to the internal one, and the opposite is done for the return value. Inter-procedural optimizations then start to propagate specialization arguments from each function call to the functional units. Constant propagation removes conditional statements and the specialization arguments themselves are removed from the function signature, as shown in Figure 3, which represents a code snippet from the implementation of rounding after the mantissa addition.

5 Experimental results

We present experimental results that compare arithmetic operators generated by TrueFloat with the ones generated by FloPoCo [2] and Template HLS [6]. Figure 4 reports the latency (in terms of clock cycles) and resources consumption (in terms of number of slices) of floating-point addition and multiplication units synthesized for a Virtex7 FPGA with 400 MHz frequency target. Five different floating-point encodings are explored: 11-bits exponent and 52-bits mantissa



(a) FPAdd

(b) FPMult

Fig. 4: Comparison between TrueFloat arithmetic units and state-of-the-art libraries in terms of clock cycles and slices consumption.

Table 1: Synthesis of TrueFloat operators with different configurations.

	FPAdd				**FPMult**				
Spec	Cycles	Slices	LUTs	Registers	Cycles	Slices	LUTs	DSPs	Registers
nih	11	280	750	961	12	216	447	10	927
nihs	12	383	979	1219	13	210	448	10	936
noh	11	259	723	869	12	206	400	10	964
tih	9	218	632	763	10	167	370	10	607
toh	9	221	610	676	10	150	316	10	674

(IEEE 754 double precision), 9-bits exponent and 38-bits mantissa, 8-bits exponent and 23-bits mantissa (IEEE 754 single precision), 5-bits exponent and 10-bits mantissa, 3-bits exponent and 4-bits mantissa. As can be seen in the plots, TrueFloat consistently delivers performance and resource usage that are competitive with respect to FloPoCo and Template HLS, especially considering that both FloPoCo and Template HLS often generate designs with half the requested frequency.

Table 1 shows double precision floating-point operators synthesized with different configurations: round to nearest even `n` or truncation `t`, IEEE compliant exception handling `i` or overflow `o`, and support for subnormal numbers (`s`). The target is a Zynq FPGA with a frequency of 200 MHz. Picking the *nih* configuration as the default, we can observe how adding support for subnormal numbers is expensive both in terms of resource usage and number of cycles, while moving from round to nearest even to truncation significantly reduces latency and area. (Note that the TrueFloat addition operation does not require any DSPs.)

We also used a larger application containing multiple IEEE 754 single precision floating-point operations (2mm kernel from the PolyBench suite) to compare TrueFloat against commercial tool Vitis HLS. Targeting a Zynq FPGA at 100

Table 2: Synthesis of the same kernel with TrueFloat (Bambu) and Vitis HLS.

PolyBench 2mm

HLS tool	Cycles	Slices	DSPs	LUTs	Registers
Bambu	65250	422	2	1022	891
Vitis HLS	76708	413	14	908	1220

MHz, the accelerator generated by Bambu has a better performance because the TrueFloat functional units allow Bambu to remove registers before and after floating-point units and save clock cycles for each operation, while Vitis HLS treats them as black boxes and cannot apply further optimizations.

6 Conclusions

The TrueFloat framework provides an automated approach to the design of multi-precision floating-point applications with customizable data types. The proposed flow can fully exploit HLS optimizations thanks to its integration within the synthesis flow, leading to improved performance and resource consumption. The compiler step and operators library are available in open source at `https://github.com/ferrandi/PandA-bambu`.

References

1. Budiu, M., Sakr, M., Walker, K., Goldstein, S.C.: BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In: Proceedings from the 6th International Euro-Par Conference on Parallel Processing (Euro-Par '00). p. 969–979 (2000)
2. de Dinechin, F., Pasca, B.: Designing Custom Arithmetic Data Paths with FloPoCo. IEEE Design & Test of Computers **28**(4), 18–27 (2011)
3. Ferrandi, F., Castellana, V.G., Curzel, S., Fezzardi, P., Fiorito, M., et al.: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In: Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC). pp. 1327–1330 (2021)
4. Rodrigues, R., Campos, V., Pereira, F.: A fast and low-overhead technique to secure programs against integer overflows. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–11 (2013)
5. Siemens Digital Industries Software: HLS Libs (2021), `https://hlslibs.org/`
6. Thomas, D.B.: Templatised Soft Floating-Point for High-Level Synthesis. In: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). pp. 227–235 (2019)
7. Wang, X., Leeser, M.: VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. ACM Trans. Reconfigurable Technol. Syst. **3**(3) (2010)