A type-safe arbitrary precision arithmetic portability layer for HLS tools

Luc Forget, Yohann Uguen, Florent de Dinechin, David Thomas

A type-safe arbitrary precision arithmetic portability layer for HLS tools

Luc Forget
Yohann Uguen
Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
Villeurbanne, France
first-name.last-name@insa-lyon.fr

David Thomas
Imperial College
London, United Kingdom
d.thomas1@imperial.ac.uk

ABSTRACT

Recent studies have shown that High-Level Synthesis (HLS) is an efficient way to design operators for floating-point arithmetic, or for emerging alternative formats such as posits. However, HLS tools support different supersets of different subsets of the C language – for example, support for arbitrary-sized bit vectors may be provided through vendor-specific data-type libraries such as `ac_int`, `ap_int`, or `int1` to `int64`, while others only support the standard C integer types. This is a problem when carefully tuning an operator's internal data-path, as there is no portable HLS standard for arbitrary width integers, and vendor libraries may introduce implicit casts and extensions that can hide subtle bugs. Each vendor also offers varying support for important operator-building primitives, such as platform-optimized leading-zero count. To address such problems, this work introduces Hint (hardware integer), a header-only compatibility layer offering a consistent and comprehensive interface to signed and unsigned arbitrary-sized integers. To avoid bugs Hint is strongly typed, requiring exact matching of expression widths and types – this type-checking is performed statically using the C++ template system, and adds no overhead at synthesis time. The current implementation wraps `ac_int` and `ap_int` with no performance or resource overhead when synthesized on Xilinx or Intel FPGAs. It also offers a Boost::multiprecision backend for fast simulation. Hint is open-source and extensible, and aims to provide an optimized superset of existing library primitives. This work is evaluated with arithmetic operators useful when implementing floating-point and posit operators (shifter, leading zero counter, fused shifter+sticky) deployed using two mainstream HLS tools (Xilinx VivadoHLS, and IntelHLS). A complete posit adder operator has also been written using Hint, showing no overhead when compared to the original operator written for Xilinx FPGAs.

ACM Reference Format:
Luc Forget, Yohann Uguen, Florent de Dinechin, and David Thomas. 2019. A type-safe arbitrary precision arithmetic portability layer for HLS tools. In *The 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2019), June 6–7, 2019, Nagasaki, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3337801.3337809

HEART 2019, June 6–7, 2019, Nagasaki, Japan

ACM ISBN 978-1-4503-7255-8/19/06…$15.00
https://doi.org/10.1145/3337801.3337809

1 INTRODUCTION

A body of recent work has shown that High-Level Synthesis (HLS) tools [6] are mature enough to implement advanced arithmetic components such as floating-point [2, 10], the emerging posit competition [7, 12], or non-standard application-specific operators [11]. When compared to a more classical HDL-based approach such as VFLOAT [13] or FloPoCo [5], this approach of implementing operators in HLS means that new operators can be added via libraries, and instantiated operators can benefit from all the high-level scheduling optimizations performed by HLS compilers.

A key advantage of implementing operators using HLS is that the operators can be both platform-independent and efficient, providing open-source and debuggable operators with similar performance to vendor IP cores. However, this performance relies on the use of highly-optimized vendor-specific libraries for custom-width integers, such as `ap_int` and `ac_int`. Floating-point and posit operators also rely on some less common integer operations, such as leading-zero counts and shifting while normalizing. These specific operations have efficient hardware implementations, but the interface as well as the implementation may vary for each target FPGA family and HLS tool. A naive implementation can increase area and reduce performance.

The API of vendor-specific libraries tends to be designed for ease-of-use, with features such as automatic sign-extension of types, and implicit conversions. Such features are useful for writing DSP applications, where padding only reduces performance. However, when implementing a floating-point operator, each bit matters, so silent padding or conversion of types may mask a logic error or un-handled corner case.

To support the development of truly cross-platform open-source custom operators for HLS, this article introduces a new library called Hint – "hardware integer" – which provides a lightweight type-safe abstraction over vendor libraries. As well as basic integer operations, it also provides optimized implementations of the more obscure integer operations needed for implementing floating-point and posit operators, taking advantage of template techniques to construct optimized data-paths and provide information to the HLS compiler. The main contributions are:

- A new open-source API called Hint, which provides a platform-independent API and strongly-typed semantics for defining custom-width integer data-paths.
- An extension API for adding new backends by defining a small number of shared operations, and a set of back-ends for widely used HLS tools.

- New compile-time optimized operators for: shifting and computing stickies; performing leading zero counts (lzc); computing combined shifts and lzc. All these operators use C++ templates to optimize each instance for the exact operand widths and shift values requested.
- An evaluation of the library using two different HLS tools (VivadoHLS and IntelHLS) with the above mentioned operators.
- The implementation of a complete posit adder (from [12]) using Hint.

The proposed API and vendor-specific back-ends are available as an open-source library at:

https://github.com/yuguen/hint

2 BACKGROUND AND MOTIVATION

2.1 Integers and HLS

The support of arbitrary-sized bit vectors is not standard in HLS tools. The nearest to a common standard is the `ac_int` templatised C++ library developed at Mentor Graphics [9]. It is supported by the commercial tools Intel HLS and CatapultC, and the academic tool GAUT. However, the HLS tool with the most traction in reconfigurable computing is probably Xilinx VivadoHLS, and it uses a proprietary library called `ap_int` [1, Ch. 4]. While `ap_int` and `ac_int` provide almost functionally equivalent support for basic arbitrary-sized signed and unsigned integers, their interfaces are different and they do not have equivalent support for operations such as leading zero count. Other tools only support widths up to 64-bits: the academic tool Augh [8] defines 64 new non-standard base types `int1` to `int64`. Two other academic tools, LegUp [3] and Panda/Bambu [6] only support the standard C integer types so code must be written with only 8-, 16-, 32-, and 64-bit integers. If only standard-width types are available, a 17-bit integer must be emulated in the code using 32-bit numbers and bit-masks – hopefully the compiler will truncate it to 17-bits during optimization, but this may not occur until late in the synthesis process, so it will be scheduled as if it were the full 32-bits.

2.2 The need for arbitrary-sized integers

Arbitrary-sized integers are extremely useful when designing custom operators: for example, for double-precision floating-point operators we have 11-bit exponents and 52-bit fractions at the inputs and outputs. Then inside an adder data-path we find a 53-bit explicit fraction and a 56-bit data after effective addition, while for multipliers, we have an intermediate mantissa product of 106 bits. This was for the standard 64-bit floating-point format, but reconfigurable computing can also take advantage of non-standard formats. Such non-standard floating-point formats have already been expressed using C++ templates [10], but these must be adapted for each HLS vendor's integer library.

2.3 Type safety

Another issue is to define the exact meaning of compound expressions involving implicit intermediate types and implicit type conversions. For instance, in the expression `(a+b)+c`, the type of the intermediate result `(a+b)` is implicit and implementation-defined. In addition, libraries such as `ap_int` and `ac_int` use operator overloading to define the types and semantics, so the behavior of the addition is defined in the library implementation, which may silently involve implicit casts if `a` and `b` have different types.

Mainstream tools such as VivadoHLS or CatapultC tend to chose the implicit intermediate types in a way that ensures that no information is lost. For instance, if both `a` and `b` are 32-bit integers, the implicit type of `(a+b)` should be a 33-bit integer to hold the possible carry out, unless the result of `(a+b)+c` is itself finally stored in a 32-bit integer, in which case all the arithmetic may happen modulo 2^{32}.

Things are a bit trickier with shifts: left shifts may or may not lose the shifted out bits. Right shifts always lose the shifted-out bits, but in the signed case they may perform a sign extension, where the size of the intermediate format will matter. The interested reader is invited to compile and run the following program:

```
#include <iostream>
int main() {
  int a,b,s;
  a = 255;

  s = 31;
  b = (a<<s) >>s;
  std::cout << b << std::endl;

  s = 33;
  b = (a<<s) >>s;
  std::cout << b << std::endl;
}
```

At time of writing, on a Linux 64-bit PC, using Clang or GCC, the first assignment to b computes -1: this can be explained by the fact that all arithmetic is performed in unsigned 32 bits. The second assignment may compute 255 (with optimization level -O0) or 0 (with -O2). This can be explained by different intermediate types for the intermediate result (a 64-bit int in the -O0 case, a 32-bit type in the -02 case). We leave it to the reader to check the generated assembly code: our main message is that the ease of use of HLS may hide subtleties that incur bizarre behaviors, but also hidden hardware or latency overheads. The "type-safe" part in the title of this work really means to give back to the designer some control of what is happening in a HLS tool.

2.4 Core arithmetic primitives for floating-point and posits

Most floating-point operators (be it IEEE-754, posit-like, or other) rely on the following basic components:

- **Arbitrary-precision addition, subtraction, multiplication**. Multiplication can be implemented out of addition, but on reconfigurable targets it can also be built by assembling DSP blocks in a clever way. Therefore, multiplication should be a primitive, and its implementation is best left to back-end tools that know the target. Division and square root can be implemented either out of addition and tabulation, or out of multiplications. Whether or not this algorithmic choice

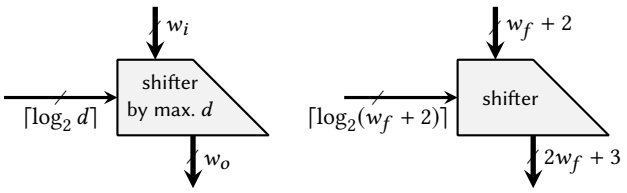

Figure 1: A generic shifter (left) instantiated (right) in a floating-point adder with w_f fraction bits

should be left to the back-end tools is out of the scope of this article.

- **Arbitrary precision shifters**. There are standard operators in C/C++ for shift operations: << and >>. As we have already observed, it doesn't mean that their behavior is defined by the standard. In a processor, we usually have shift instructions that input the shift value and an integer, and output an integer of the same size (with possible loss of information). The C shift operators expose these instructions. Now if we are generating hardware, it is interesting to generalize as depicted in Figure 1: a shifter may be defined by an input width w_i, a maximum shift distance d, and an output width w_o. The shift input will be an integer on $\lceil \log_2 d \rceil$ bits. The shifter can be errorless (no shifted-out bits) if $w_o \geq w_i + d$.
- **Arbitrary precision leading-zero counter, leading-one counter, and leading-bit counter**. The latter counts ones if the leading (leftmost) bit is a one, and counts zeroes if the leading bit is a zero.

2.5 Fused arithmetic primitives

To reduce the data-path width and improve the delay, it is often useful to merge these operators:

- A **multiply-add** computes $A \times B + C$.
- **Addition with carry-in** is in principle no more expensive than plain addition. Writing $a + b + 1$, or $a + b + c$ where one of the three variables is a single bit, should translate to a single adder.
- A **shifter-sticky** is a shifter whose output size is the same as the input size. Therefore, bits may be shifted out. This operator incrementally computes the logical OR of all the shifted-out bits (historically called sticky bit). This slightly saves on the latency, and more importantly saves the hardware that would have shifted further all these bits.
- A normalizer is a **combined leading-zero counter and shifter**. It is a leading-zero counter that, at the same time, shifts the input so that the leading bit of the output is the first non-zero bit. It outputs the normalized result, along with the number of zero bits that were counted. Posits make use of a similar combined leading-bit-counter and shifter.

2.6 Support of these primitives in HLS tools

All the tools support those primitives which are part of C (addition, multiplication, shifts), although the actual behavior may be implementation-defined in some cases. Sometimes, they are implemented as highly optimized IP cores. Sometimes, they are implemented as libraries.

The most notable missing basic operation is the leading bit count. Among the fused operations, only the add with carry-in is sometimes supported.

3 TYPE SAFETY FOR ARBITRARY-PRECISION INTEGERS IN HLS

Current HLS integer libraries perform very little compile-time sanity checks. This section describes a Hint variable declaration and its elementary methods along with their type-checking. These basic methods are used to build more complex operators.

3.1 Variable declaration

The Hint library is a templated wrapper above underlying backends. A user must write it's function as a template function to be able target all the backends. In that case, it's function will have as template parameter `template<unsigned int , bool> class HintWrapper`. This defines a type that a user can use to declare it's variable: `HintWrapper<W, s> var;` defines a variable `var` on `W` bits and has a sign `s`.

3.2 Variable assignment

In vendor tools, assignment to a variable of mismatched size is a cause of silent truncation and potentially unused bits. The proposed library diverges from `ac_int` and `ap_int` as it only allows assignments of identical size variables. Furthermore, it only allows for assignments with matching signedness. The variable assignment is performed by using the usual "=" operator.

This restriction requires the programmer to explicitly truncate their variables when a part is not needed. A compiler error will be thrown if these properties are not ensured. Enforcing this behavior helped us discover several bugs in our operators.

3.3 Slicing

As a preliminary note, it is interesting to remark that the slicing methods for `ac_int` and `ap_int` have different restrictions, and may therefore not be interchangeable. In `ac_int`, a bit slice of size `S` starting as bit weight `l` of the variable `var` is written `var.slc<S>(l)`. Conversely, the `ap_int` slicing method is `var.range(h,l)` where `h` and `l` are the weights of the MSB (most significant bit) and LSB (least significant bit) from which to slice `var`.

The difference is that the value of `S` must be known at compile time. Therefore, a slice whose size varies in a loop will compile using `ap_int` but not using `ac_int`.

Neither of these two libraries is able to check at compile time if the slice is out of bounds. By having `S` as a template parameter, `ac_int` ensures that the size of the output is known at compile time. However, if the user assigns the result of the slice to a larger or smaller variable, the result might be truncated without warning. When using `ap_int`, no compile-time checks can be performed. The result of an out-of-bound slice will have some of its bits set to 0.

For Hint, we choose to be even more restrictive in order to allow checks at compile-time that slices are in range. A `Hint` slice is of the form `var.slice<h,l>()` where `h` and `l` are the weights of the

MSB and LSB of the slice of `var`. As `h` and `l` are known at compile time, sanity checks are performed and the output size is known. Therefore the program won't compile if `h<l` or `l<0` or if `h>=W` with `W` being the size of `var`. The size of the returned integer is `h-l+1`. It cannot be truncated implicitly, as we have seen that assignments are only allowed between matching sizes.

3.4 Concatenation

Both libraries are able to know at compile time the size of the result of a concatenation. However, there may still be silent truncation when assigning this result to a smaller variable. The proposed concatenation method is `var1.concatenate(var2);` where the result is of size of `var1` + the size of `var2`.

3.5 Others

The Hint API can be extended with any other methods with the same spirit that all types must be checked at compile time. For example, the current implementation contains:

- bitwise operations such as and, or, xor that from two identical W width variables returns a W bits variable containing the corresponding bitwise operation:
  e.g. `var1.bitwise_and(var2)`
- and/or reductions that returns a single bit result:
  e.g. `var1.and_reduction()`
- a signal inverter that transforms each bit to it's opposite:
  `var1.invert()`
- an operator that computes the reverse of a variable (the lsb takes the msb and so on): `var1.backwards()`
- a padding operator that performs the extension of a Hint variable to a larger one; only available if is result size is larger than the original size: `var1.leftpad<newsize>()`
- a generator of a sequence of a given length containing identical bits: e.g. `HintWrapper<W, s>::generateSequence(bit);` where `bit` is a single bit to be replicated.
- a sign inverter: `var1.invert()`
- an equality operator `"=="` that only compares identical width and sign Hint variables
- a multiplexer operator that takes a control bit and two identical width variables: `mux(control, var1, var2)`
- a modular addition (`var1.modularAdd(var2)`) that performs the sum on two `W` bits variables and return the sum of these two values on `W` bits when the user knows the addition won't overflow. Similarly, Hint also provides a modular subtraction (`var1.modularSub(var2)`)
- a adder with carry that takes two `W` bits variables with a `1` bit carry and returns a `W+1` bits sum of the three:
  `var1.addWithCarry(var2, carry)`

All the types are safely deduced by the compiler using the `auto` keyword as each operator returns a specific type depending on its inputs.

This list is subject to grow, following applications needs, hence requiring each backend to implement these basic functionalities. However, this set of basic functionalities allows for building higher-level operators.

4 PORTABILITY TO MAINSTREAM HLS TOOLS

There are several ways to implement vendor-specific back-ends for the proposed interface. This section presents three approaches, each with their pros and cons.

4.1 Class duplication for each backend

In order to have compile-time decision of which backend to implement, one can simply enable/disable the Hint class definitions depending on environment variables (using `#if`-based conditional compilation) . This implementation is the less elegant way of implementing such a portability layer. It is also the most error prone: bit-for-bit portability relies on each backend implementing the same methods and the same static verification semantics, despite sharing no code and compiling to different libraries.

This poor engineering approach was nevertheless used to check the feasibility of a portability layer. A complete posit adder implementation initially written for `ap_int` was ported to Hint ([12]). When compiled to VivadoHLS, no degradation of the quality of results was observed. Meanwhile, the posit operator could now be compiled with IntelHLS.

4.2 A shared class interface

The approach of using a conventional interface that each backend follows is a bit more elegant. There is an implicit interface that each backend must implement in order for the operators built upon to compile. This is also true regarding the static verification which are duplicated in each backend. Thus two backends might not perform the same static verification.

4.3 Curiously recurring template pattern (CRTP)

An elegant way of centralizing the static checks is to use a CRTP class [4]. A front class is provided to the user for instantiating a custom hint. It is templatised by a width, a sign and, a backend. A given backend inherits from the associated specialised Hint class. Therefore the Hint class is the frontend of the library. It implements all the API methods and is in charge of performing the static verification. If such verification are satisfied, a call to the underlying backend implementing the same method is issued.

This approach allowed for correct software simulations for both VivadoHLS and IntelHLS. Indeed, this approach only uses features from C++11. Unfortunately, synthesis results showed here that both tools were unable to pipeline complex operators in this case. Further investigations showed that a templated hint function, when implemented as a CRTP, results in a monolithic block that cannot be pipelined. The recursive template calls insert registers that the optimizer is unable to remove, resulting in a high latency, resource hungry operator.

We expect that future versions of the vendor tools will catch up. In the meantime, the second approach will be used. There is also some longer-term hope that "concepts" introduced in C++20 will make CRTP useless.

Table 1: Synthesis of lzc and shifters on Arria 10 (achieved clock target of 240MHz)

	N	ALMs	FFs	MLABs	cycles
native type lzc	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8
hint type lzc	26	32.5	32	0	1
	55	86.5	91	1	5
	256	465.5	710	1	8
hint lzc + native shift	28	93	106	0	2
	57	218.5	213	0	2
	64	296.5	340	0	3
	256	1487	1238	13	7
	279	1603	1322	14	7
hint lzc + shift	28	88.5	72	0	1
	57	212	209	0	2
	64	279.5	308	0	3
	256	1388	1960	0	6
	279	1455.5	1544	0	4

Table 2: Synthesis of lzc and shifters on Kintex 7 (achieved target delay of 3ns)

	N	LUTs	FFs	SRLs	cycles
builtin lzc	26	50	81	0	4
	55	85	111	0	4
	256	475	559	11	9
native type lzc	26	26	57	1	4
	55	68	87	10	5
	256	233	516	5	7
hint type lzc	26	26	57	1	4
	55	69	87	11	5
	256	234	516	5	7
hint lzc + native shift	28	96	81	0	8
	57	222	144	0	9
	64	264	142	0	8
	256	1532	1045	0	11
	279	1691	1131	0	12
hint lzc + shift	28	102	122	0	4
	57	254	297	0	5
	64	292	275	0	6
	256	1164	1568	0	8
	279	1958	2265	0	10

5 EVALUATION

All the presented results are given after place-and-route. VivadoHLS 2016.3[1] was used when targeting Kintex 7; IntelHLS 19.1 when targeting Arria 10.

The evaluation is divided in four parts: ensuring that no overhead is generated, implementing combined operators that reduce resource consumption, then latency, and demonstrate the Hint library a larger project.

The overhead evaluation is performed on the implementation of a lzc. The lzc algorithm chosen in this paper has been implemented using `ac_int`, `ap_int` and Hint. The synthesis results are given in Tables 1 (top) and 2 (top) for Intel and Xilinx respectively. VivadoHLS provides a builtin lzc, which is also presented here. The sizes (N) of the inputs corresponds to real world examples. Indeed, 26 and 55 bits are the width of the lzc needed in single and double precision floating-point adders while a 256 bits lzc is needed for a 32 bit posit quire.

The comparison between the native type implementation and the hint type implementation shows no overhead when using Hint. Furthermore, the implemented lzc algorithm outperforms the VivadoHLS builtin lzc both in term of resources and latency.

The first combined operator presented is the shifter+sticky; reducing resource consumption. The shifted out bits are not discarded, but ORed in a "sticky" bit. The fused operator attempts to OR these bits inside of the shifter, before they are shifted out. This saves the logic that otherwise shift these bits to their final place before the final wide OR.

The synthesis results of the shifter+sticky are presented in Tables 3 and 4. The sizes (N) of the operators comes from the floating-point adder in single (27 bits) and double (56 bits) precision. As both tables show, this optimization saves a considerable amount of logic on both targets. Table 3 does not report the number of cycles required for said operators. Indeed, IntelHLS was giving untrustworthy latency results. However, the circuits were cosimulated to ensure that they produced the correct mathematical results using ModelSim.

Table 3: Synthesis of shifters+stickies on Arria 10 (achieved clock target of 240MHz)

	N	ALMs	FFs	MLABs
native shift + sticky	27	134	63	2
	56	277	212	3
hint shift + sticky	27	82	40	0
	56	179.5	128	0

Table 4: Synthesis of shifters+stickies on Kintex 7 (achieved target of 3ns)

	N	LUTs	FFs	Cycles
native shift + sticky	27	113	110	3
	56	309	234	4
hint shift + sticky	27	84	65	3
	56	203	133	3

The second combined operator presented is a lzc+shifter; reducing latency. A lzc is usually followed by a shift as mostly used in floating-point or posit normalizer. Combining these two operators allows to reduce the latency of the design. Indeed, both the shift and the lzc are divided in stages where one can an lzc step as well as a shift step. This removes the data dependency of the complete lzc computation before issuing the shift at the expense of more logic.

[1]This older version of VivadoHLS is used because 2018.3, the latest version, at the time of writing, proved very unstable, with numerous crashes and sometimes silent production of incorrect hardware. VivadoHLS 2016.3 is the best compromise between stability and quality of results in our case.

To evaluate such an operator, a combined hint lzc+shift is compared to a hint lzc followed by a native shift (>>). Tables 1 (bottom) and 2 (bottom) provide synthesis results of these implementations for both Intel and Xilinx FPGAs. In addition to sizes previously presented for the lzc, we added the sizes of a 16 bits quire normalizer (64 bits) and of a 32 bits Kulisch accumulator normalizer (279 bits).

For both Intel and Xilinx, the latency of the combined lzc+shift is improved compared to a serial implementation. In some cases, the tools are even able to reduce resource consumption. This might be due to them being able to compress multiple stages of lzc+shift in a better way than with a separated design.

Finally, a complete posit adder [12] has been rewritten using Hint without overhead compared to the original version written using `ap_int`. The architecture involves the use of two lzc+shift and two shifters+stickies as long as additions, comparisons, logic functions, concatenation, etc.

6 CONCLUSION

This work provides an open-source portability layer for custom-size integer datatypes called Hint. It is strongly typed: no information is lost or useless bits appended when performing operations on Hints without explicit programmer request. This is enforced at compile time with static type checks from the C++ compiler. These static checks follow a well defined semantic, making every operation explicit about the types it manipulates. This may sound very restrictive, but ultimately, most Hint variables can safely use the `auto` C++ type, as all the widths and signedness are derived and checked from the inputs.

The Hint library allows one to write a single operator that can be synthetized using different vendor platforms. The component can also be efficiently simulated using the Boost::multiprecision backend. Using Hint, the computation results are guaranteed to be identical on every platform.

Hint does not induce any overhead when using mainstream HLS tools compared to their native types implementations.

In general, work on this project has been considerably slowed down by vendor tools limitations. They are currently unable to equate identical variable through the template layers involved by the best implementation, although the programs are accepted and properly simulated within the tools. In addition, depending on the C++ construction used, two functionally equivalent programs can result in drastically different synthesized hardware.

These issues are being understood and ironed out. The wider goal is to build a larger collection of sophisticated arithmetic operators upon this set of trusted operations. The first step towards this will be to complete the port of existing open-source HLS works such as floating-points [10] or posit [7, 12] operators. The next step will then be to apply the same concepts to fixed-point formats (wrapping `ac_fixed` and `ap_fixed`).

A longer-term objective is to build on the clear semantic of every basic operation to build formal proofs of the correct behavior of the hardware.

REFERENCES

[1] Vivado Design Suite User Guide: High-Level Synthesis (UG902). https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf, 2018.

[2] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson. High-level synthesis of software-customizable floating-point cores. In *2018 Design, Automation & Test in Europe*, pages 37–42. IEEE, 2018.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA international symposium on Field-Programmable Gate Arrays*, pages 33–36, 2011.

[4] J. O. Coplien. Curiously recurring template patterns. *C++ Report*, 7(2):24–27, 1995.

[5] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[6] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[7] A. Podobas and S. Matsuoka. Hardware implementation of POSITs and their application in FPGAs. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 138–145. IEEE, 2018.

[8] A. Prost-Boucle, O. Muller, and F. Rousseau. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 60(1):79–93, 2014.

[9] A. Takach. Algorithm c (AC) datatypes. https://github.com/hlslibs/ac_types, 2018.

[10] D. Thomas. Templatised soft floating-point for high-level synthesis. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019.

[11] Y. Uguen, F. de Dinechin, and S. Derrien. Bridging High-Level Synthesis and Application-Specific Arithmetic: The Case Study of Floating-Point Summations. In *Field-Programmable Logic and Applications*. IEEE, Sept. 2017.

[12] Y. Uguen, L. Forget, and F. De Dinechin. Evaluating the hardware cost of the posit number system (Online). 2019.

[13] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *FCCM*, pages 249–258. IEEE Computer Society, 2006.