Templatised Soft Floating-Point for High-Level Synthesis

David B. Thomas
Department of Electrical and Electronic Engineering
Imperial College London
Email: dt10@imperial.ac.uk

Abstract—**High-level Synthesis (HLS) tools have greatly increased the productivity of FPGA application development, making it possible to easily create highly-parallel application-accelerators. However, while FPGAs are known for the ability to customise the number representation of data-paths, most HLS work only uses custom-precision for fixed-point representations, and for floating-point relies on the 64-, 32-, and 16-bit formats provided by vendors. This paper presents a solution for parametrised floating-point in HLS via C++ templates, allowing for compile-time selection of exponent and fraction widths, including the use of mixed precisions for input arguments and result types. By using arbitrary width integers and compile-time logic the resulting operators describe the same data-path as an external floating-point IP generator, while still allowing the HLS tool to perform detailed optimisation and scheduling of the internal components. We show that the resulting custom-width HLS cores provide similar area and performance to platform-native vendor IP blocks, while adding full support for heterogeneous precision floating-point data-paths to HLS tools.**

I. Introduction

Contemporary FPGAs offer a large amount of fine-grained logic plus many integer DSP cores which can be used to construct application-specific Floating-Point (FP) data-paths. Traditionally these data-paths were constructed using Register Transfer Level (RTL) descriptions, but the increasing scale and complexity of applications means that High-Level Synthesis (HLS) tools are now often used to describe applications in C-like languages. Modern HLS tools incorporate built-in support for single- and double-precision FP types and operations, hiding the complexity of scheduling and binding from the programmer, and instantiating high-performance device-specific FP IP blocks on demand.

In RTL it is relatively easy to specify custom-precision FP operations, and it is known that carefully optimising FP types can reduce resource usage while maintaining accuracy. However, FP support in HLS tools usually only offers a limited set of FP types, as FP cores are now black-boxes that are opaque to developers. The scheduling of monolithic pipelined blocks may also limit scheduling efficiency – though HLS schedulers have become better at intra-cycle scheduling of combinatorial components, the IP block's internal pipeline registers may still lead to scheduling of IP blocks at a per cycle level.

This research was supported by RAEng/The Leverhulme Trust Senior Research Fellowship grant LTSRF 1718\14\38.

This paper argues that modern HLS tools are sophisticated enough that FP operations can be brought in-house: rather than treating FP blocks as monolithic IP blocks, the programmer can instead describe them as fine-grain data-paths, opening them up to the scheduler. This idea goes beyond the classic approach of creating software-defined FP for specific precisions, as we show that there are opportunities and advantages to creating templated C++ libraries that are able to implement any heterogeneous combination of input and argument precisions. The programmer can then choose from thousands of possible configurations to match the needs of their application. A key enabling technology is the C++11 front-end of modern HLS tools, which allows us to embed the code generation intelligence of a library such as FloPoCo into a parametrised synthesisable C++ library.

The main contributions of this paper are:

- The idea of templatised soft FP operators, which allow programmers to specify heterogeneous FP types in their source code, and then have the requested customised operators generated at HLS compile-time without needing any extra tools.
- A concrete implementation of this idea, in the form of a platform-independent open-source library allowing for heterogeneous FP designs in commercial HLS tools: https://github.com/template-hls/template-hls-float.
- An evaluation of the library using Vivado HLS, which shows that templatised soft FP cores can offer similar or better performance and resource efficiency to vendor IP cores, and that in composite data-paths they can actually beat vendor cores in terms of accuracy versus resources.

II. Background

FP representations are designed to preserve relative accuracy over a wide dynamic range, as opposed to fixed-point representations which are able to maintain absolute accuracy over a narrower range. A binary FP number is usually split up into: an exponent $e \in \mathbb{Z}$ which determines a specific binade (or scale); a fraction $f \in [1, 2)$ which identifies a number within the binade; and a sign $s \in \{-1, +1\}$. Because the fraction always starts with a leading one, a reduced fraction $f' \in [0, 1)$ is usually stored, then combined with an implicit 1 when needed. The represented value v is then defined as:

$$v = s \times 2^e \times (1 + f') \qquad (1)$$

Practical FP representations must choose concrete fixed widths for e and f; using more bits for e increases the dynamic range, while using more bits for f increases the precision. Most applications need a way to represent zero, and in the general case it is important to define infinity and Not-a-Number representations too.

The IEEE 754 FP standard is the most popular FP format, which defines a portable standard for the representation of FP numbers, and also requirements on mathematical operators and rounding [1]. The standard defines three "basic" binary representations, for single (32-bit), double (64-bit), and quadruple (128-bit) numbers, but it also defines "interchange" formats for 16-bit representations and for all multiples of 32-bit. The standard can also usefully be interpreted to implicitly define semantics and representations for any combination of exponent and fraction widths.

While standard IEEE has been extremely successful in software, other FP formats have been found useful. For example, some AMD GPUs use a 24-bit (We=7,Wf=16) FP format internally [2], while the bfloat16 (We=8,Wf=7) has recently been used in a number of machine-learning applications [3], and Intel has now defined a standard form due to it's expected importance across its platforms [4]. Non-IEEE FP has a long history in FPGAs, where it is possible to omit the complicated steps needed for denormal numbers as they add significant latency and resources [5]. FPGA application designers have also explored the customisation of FP types in applications, either by choosing custom FP operators [6], or by varying the exponent and fraction widths within a data-path to match numerical properties of an application [7].

Given the need for FP operators in FPGA designs, a number of different sources for FP IP are available. The first route is by vendor-supplied IP core generators, such as Xilinx [8] and Intel [9] FP libraries. These are proprietary black-box IP generators, which can produce FP IP blocks optimised for a user-chosen combination of FP type, target platform, and target pipeline depth. There is also some support for automatically choosing a pipeline depth for a given target clock, based on up-front frequency characterisation by vendors.

The FloPoCo tool is an open-source and platform-independent alternative to proprietary vendor tools, and generates non-encrypted VHDL operators rather than opaque IP cores [10]. FP operator architectures are converted to C++ code, and when the C++ code is run it uses FP type information specified by the user to output a specialised VHDL IP core. FloPoCo attempts to adjust pipeline depth in order to meet a specific target frequency, using an internal model for delay characteristics of different FPGAs. Because FloPoCo does not have access to proprietary FPGA knowledge and cannot specialise as much for each architecture it is at a disadvantage compared to proprietary IP generators, but in practise the area-delay performance of FloPoCo IP cores is competitive to platform-specific vendor cores.

Another source of FP IP comes from parametrised RTL code, which provide FP operator entities with generic inputs describing FP types, with data-path widths determined at elaboration time. The best known example is VFLOAT [11], which provides a number of operators and has been actively maintained and used for over a decade across many FPGA architectures and generations. The parametrised RTL approach is similar to the templatised soft IP approach proposed here, though we work within C++ and rely on the HLS scheduler to perform timing. There are also limits to the parametrised RTL approach due to the difficulty of writing correct generic VHDL with multiple code-paths and doing computation at compile-time; the authors of FloPoCo originally provided a parametrised IP library called FPLibrary [12], but switched to C++ generation via FloPoCo in order to allow more specialisation and compute during IP generation.

C-like HLS tools existed before the popularity of FP on FPGAs, and many early HLS tools did not natively support `float` and `double` as they were seen as too expensive. However, as FPGAs got bigger ad-hoc software-defined solutions were used to implement FP; an early academic example of C-defined FP appears in 2001 [13], while Handel-C provided a commercial C-defined FP operator library in 2005 [14] which supported a defined set of standard and non-standard precisions. There is also a long history of software-defined FP for CPUs using C, such as the SoftFloat library [15], which can be used both as a reference FP implementation and also to provide FP support in embedded CPUs lacking a co-processor. Soft floating-point libraries have also been developed for HLS tools, such as [16], though these do not support the compile-time generation of IP cores, and so only support pre-chosen operator widths.

Despite the rich history of custom FP in FPGAs, there are currently few solutions for using custom FP operators in HLS, beyond implementing software defined libraries for fixed-point number representations. One related approach considers the use of HLS to build non-standard floating-point operations such as summation [17], which extends the set of available primitives. However the goal in this paper is to replace the set of primitives, and use the C++ front-end of modern HLS tools to provide a solution that can support a heterogeneous mix of number representations, both within each application, and also within each FP expression.

III. TEMPLATISED SOFT FLOATING-POINT

An ideal FP library for HLS would provide the flexibility of a tool such as FloPoCo or a library such as VFLOAT, providing the ability to customise data-path logic based on the types of expressions. At the same time it would ideally be integrated directly into the scheduling engine of the HLS tools, which can provide device-specific context and accurate intra-cycle timing information. Our solution is to use C++ templates to implement the data-path generation logic directly in the C++ library, generating the core at C++ elaboration time. This approach means the synthesis back-end receives a graph of precisely-sized fine-grain operations, which it can then schedule according to detailed timing knowledge.

We will first sketch out a simple version of this approach to show the main ideas, then later describe the specific details chosen to create a real library. As an example we'll use a multiplier, as it is the simplest of the operators.

First we define a concrete FP representation to describe numbers within the program. This must be templatised with exponent and fraction parameters, describing the widths, and assumes we have some variable width integer library `wint<W>`:

```
template<int We, int Wf>
struct fp
{
  bool neg;      // If true the number is negative
  wint<We> exp;  // Biased exponent
  wint<Wf> frac; // Fractional part with implicit leading 1
};
```

we can then create instances of the type:

```
fp<8,23> fval; // Equivalent to IEEE single precision
```

Operations on these types can then be defined as functions which take and return parametrised types defining the types of expressions. Existing solutions for custom FP define one precision per operation, leading to a multiplication function with the following signature:

```
template<int We, int Wf>
fp<We,Wf> void mul_std( fp<We,Wf> a, fp<We,Wf> b);
```

However, this is overly limiting as it forces an approach where input arguments must be forced to the more precise type. It also means that precision cannot be preserved, particularly in multiply-accumulate situations: a multiplication of two `Wf`-bit fraction numbers can produce more than `Wf` bits of precision, and if the accumulator is higher precision it can incorporate the extra precision.

A more flexible approach is to provide true heterogeneity in the types of expressions, allowing different input and output precisions to be freely mixed. The correctly rounded result is well defined, so the user can customise the widths according to their knowledge of the true precision of the numbers. This approach gives us a signature as follows:

```
template< int WeR,int WfR, int WeA,int WfA, int WeB,int WfB >
fp<WeR,WfR> void mul( fp<WeA,WfA> a, fp<WeB,WfB> b)
```

While this explicit approach may seem verbose, it provides the most flexibility to users, and in practise one only needs to specify the output precision:

```
auto v = mul<7,17>(x,y);
```

The variable `v` will have type `fp<7,17>`, and the input precisions (`WeA`,`WfA`,`WeA`,`WfA`) will be inferred from `x` and `y` by the C++ frontend.

To illustrate the advantages and problems encountered in a templatised soft-float example, we'll briefly examine a standard FP multiplier, shown in Figure 1. Note that this multiplier omits many special cases, but it demonstrates the four main stages of multiplication:

1) Use a `(1+WaF)*(1+WbF)` fixed point multiplier to form the product fraction, and form the exponent as the sum of the input exponents minus the argument-specific exponent bias.

```
template<int WrE,int WrF,int WeA,int WfA,int WeB,int WfB>
fp<WeR,WfR> mul( fp<WeA,WfA> a, fp<WeB,WfB> b)
{
  // 1 - Form the raw product components
  auto frac=((1<<WeA)|a.frac)*((1<<WeB)|b.frac);
  auto neg =a.neg ^ b.neg;
  auto exp =(a.exp-(1<<WfA)) + (b.exp-(1<<WfB));

  // 2 - Normalise the fraction to [1,2)
  if( get_msb(frac)==1 ){
    frac = frac>>1;
    exp  = exp+1;
  }

  // 3 - Rounding the number
  auto rbit = rounding_bit(frac)
  auto frac_rnd = truncate(frac) + rbit;

  // 4 - Package the components up for the next user
  return fp<WeR,WeF>{neg,exp-(1<<(WeR-1)),frac_rnd};
}
```

Fig. 1. Toy multiplication example.

2) The fixed-point multiplier produces a number in the range $[1, 4)$, so if necessary we need to normalise back to the range $[1, 2)$ with a shift by one bit.
3) The full product has `2+WaF+WbF` bits, so we need to round back to the target width of `1+WrF` bits. For simplicity we hide many of the details here, as this area is quite complicated – for example, the exponent might need adjustment in cases of overflow.
4) Packaging up the final results, including biasing the exponent, and dealing with various issues related to NaN, zeros, and infinities (not shown).

Many of the details are hidden here, with most of the code related to edge-cases in rounding and special-case handling. However, it is ultimately just normal C++ code, and after C++ elaboration it will look to the synthesis back-end like any other sequence of operations.

The difficulty comes in the details, with a number of practical issues that must be handled even in the code sketch:

Intermediate types: The code above makes extensive use of C++11 `auto`, but in reality each of the types must be carefully sized in order to make sure that information is not lost. This is particularly important with heterogeneous expressions, where all the exponent and fraction widths could be different.

Expression mappings Ultimately this sequence of expressions needs to map to LUTs and DSPs, so we must try to make sure that each expression is presented to the synthesis tool such that it will infer the "correct" resource.

Alternative code-paths When handling heterogeneous types the code-path needs to change in response to the input widths: for example, we may have the case that `1+WfR >= 1+WfA+1+WfB`, in which case rounding logic is no longer needed, and output padding must be added instead.

These problems are particularly clear when one tries to write a single function which can handle all cases, rather than writing different functions which each support a different width: it is relatively easy to write C++ which models the RTL output of an external IP generator for one parametrisation, but much more challenging to embed the data-path specialisation logic of the IP generator to allow for any parametrisation at compile-time.

A. *Expected Advantages*

This paper argues for the overall idea of templatised soft FP, beyond the one implementation presented here. Based on the sketch we have just seen, we can now argue for some of the expected advantages we expect to see.

Custom FP types: The most obvious advantage of this approach is that all FP bit-widths become accessible to the HLS programmer, allowing them to tune bit-widths according to accuracy, performance, and storage needs. Most FP HLS work has been performed in double, single, or half precision, but there is ample evidence from RTL designs that other precisions are extremely useful.

Heterogeneous precisions: Homogeneous precision FP operators still need to promote different input types to the most precise shared type, then cast the result back to the desired type, wasted area due to excess precision and introducing double-rounding. For example, when calculating the dot product of a vector of `half` and vector of `float`, we either need to implement the multiplication as a full single precision multiplier:

```
float dot_accurate_big( hls_half x, float y )
{ return ((float)x) * y; }
```

or we can round from single to half and do a half precision multiplication:

```
float dot_inaccurate_small( hls_half x, float y )
{ return x * ((hls_half)y); }
```

Using heterogeneous types allows us to access a mid-way point which retains full accuracy, but requires a smaller 11 x 24 bit fixed-point multiplier internally:

```
fp<8,23> dot_accurate_medium( fp<5,10> x, fp<8,23> y )
{ return mul<8,23>( x , y ); }
```

Increased accuracy: Heterogeneous precisions can also produce a more accurate result with little overhead. A common pattern is to perform multiply-accumulates and/or dot-products where the accumulator is wider than the input arguments, for example in matrix multiplication, convolution, or machine learning. Using homogeneous types we could either promote the arguments to the multiplications, resulting in more accuracy but also much more area:

```
double dot2_accurate_big(float x[2], float y[2])
{ return double(x[0])*double(y[0])+double(x[1])*double(y[1]); }
```

or we could do the multiplications in single precision, then promote the two partial products to single precision before adding:

```
double dot2_inaccurate_small(float x[2], float y[2])
{ return double( x[0]*y[0] ) + double( x[1]*y[1] );
```

However, the full product must have been constructed internally during the single-precision multiple, and with heterogeneous types we can access it at almost no hardware cost:

```
fp<11,52> dot2_accurate_small(fp<8,23> x[2], fp<8,23> y[2])
{return add<11,52>(mul<8,52>(x[0],y[0]),mul<8,52>(x[1],y[1]));}
```

This approach uses the same number of DSP blocks as the second method, while providing the accuracy of the first.

Portability and testing: Strict IEEE-compliant FP provides portability between devices, but often FPGA FP cores are non-strict to reduce cost and some use alternative representations such as FloPoCo floats for more efficient implementation, so outside of vendor environments it is difficult to replicate bit-exact behaviour. Software specified FP allows for bit-exact matching between reference software in C++ and hardware outputs – any deviation means there is some kind of synthesis error.

Improved scheduling: Monolithic pipelined FP IP blocks are scheduled as an indivisible unit, and cannot be moved freely within the schedule due to their internal registers. In contrast, a soft FP core exposes all it's component operations to the scheduler, meaning they can be freely moved around within the schedule, including across basic blocks. In some FP operations the fraction becomes ready earlier than the exponent, so any following operations can start work on the output fraction before the exponent is known.

IV. Approach

This paper presents one possible implementation of the templatised soft FP idea, and there are many opportunities to improve on this solution. However, it does result in a portable and efficient open-source C++ library which supports variable precision addition, multiplication, division, comparison, and float to fixed conversion, and so covers many of the low-hanging use-cases for FP in HLS. We will now present the design decisions taken in light of the potential advantages we wanted to explore, and also the expected problems.

A. *Base algorithms*

Correctness of implementation is a key concern, so an existing open-source FP IP generator or parameterised library was needed as a base to provide proven data-paths. Two clear candidates are FPLibrary [12] and VFLOAT [11]. Here VFLOAT has the advantage of an IEEE compliant representation with denormals plus active maintenance, though the custom representation used in FPLibrary (the precursor to FloPoCo format) has some advantages. However, both these libraries assume a homogeneous type for operators, rather than the heterogeneous types desired here.

The FloPoCo IP generator does partially support heterogeneous types internally, even though they are not supported on the command-line interface. This support is incomplete, but does support the most interesting use-case of heterogeneous fraction widths for multipliers. Starting from a code generator is also better aligned with the concept of templatised soft FP, as FloPoCo generators sometimes generate different paths for different combinations of widths, rather than simply adjusting the size of data-paths operations. FloPoCo inherits it's FP format from FPLibrary, which means there are two extra flag bits encoding whether a number is 0, "normal", infinite, or NaN. This separation means that special-case handling is somewhat removed from the exponent and fraction calculation data-path, potentially reducing latency and increasing the potential for scheduling freedom we hope to see. Finally, FloPoCo is a well optimised and tested library which has seen wide use, so we have a reasonable expectation that the base library is correct and reasonably performant.

As a result, we chose the FloPoCo code generators and FP representations as the base for our templatised soft FP library. The canonical representation of an FP value is given by the following struct (we will discuss the `fw_uint<W>` type later):

```
template<int We, int Wf>
struct fp_flopoco{
  const fw_uint<2+1+We+Wf> bits; // Underlying storage

  fp_flopoco(fw_uint<2> _flags, fw_uint<1> _sign,
             fw_uint<We> _exp, fw_uint<Wf> _frac)
    : bits(concat(_flags,_sign,_exp,_frac))
       {}
  ...

  // Accessors
  fw_uint<2> get_flags() const
  { return get_bits<1+1+We+Wf,1+We+Wf>(bits); }

  fw_uint<1> get_sign() const
  { return get_bit<1+We+Wf>(bits); }
  ...
};
```

Rather than representing each FP field as a field in the struct (as seen in the earlier example), a single bit-vector is used internally. In Vivado HLS this has the advantage that the FP components aren't turned into individual ports at the synthesised interfaces, and experimentally it seems that it is able to schedule the individual bit segments separately.

The `fp_flopoco<We,Wf>` type only manages data representation, so all operations are implemented as separate functions, which means we can have many different implementations of operators. Instances of `fp_flopoco<We,Wf>` can be constructed from bits in synthesised code, or there are also debug functions allowing construction from numbers from the MPFR arbitrary precision library. The only functions provided by the carrier type are accessor member functions used to access different fields such as the flags, sign, exponent and fraction, and utility functions used to extract the number as MPFR or double-precision types at run-time. Because each instance should represent an entire number, there are no modifier functions and each instance is immutable after construction;

B. Variable width integers and type-safety

Efficient FP types require precisely sized intermediate bit-vectors, as too wide a bit-vector incurs extra cost, while too narrow a variable might truncate important bits. Some tools provide custom integers, such as the System-C `sc_uint<W>` and Vivado HLS's `ap_uint<W>`, which explicitly indicate the width of types. In other tools such as Legup one must choose the narrowest width from the standard `uint{8,16,32,64}_t` types, then use bit-wise masks to make clear that upper bits are zero.

A goal of this tool is portability, so rather than choosing an existing type or emitting bit operations, a new facade type called `fw_uint<W>` was created. This type currently has three backends that can be used at compile-time:

- `ap_uint` : Wrapper for a `ap_uint`.
- `masked_uint` : The `fw_uint` contains the smallest standard integer able to hold W bits, and uses mask operations after every operator.
- `cpp_uint` : The `boost::multiprecision::number` library provides arbitrary width integers in software.

This allows FP operators to be written in a platform independent way, while hopefully maximising the performance and minimising resources on each platform. The `fw_uint<W>` type is relatively small, with helper types such as `concat<W...>`, `get_bits<Hi,Lo>`, `take_lsbs<Lo>` and so on written as platform-independent free functions.

Beyond providing platform independence, having an intermediate type was found to be critically important for correctness. Initial attempts to use `ap_uint<W>` to implement FP operators failed, as the operators would compile, but then took a very long time to debug and verify for many parameter sets. Ultimately this was down to the "helpful" casting and conversion provided by user-facing types like `ap_uint<W>`, as width mis-matches turned into silent conversions at compile time, and were not visible unless tested.

To turn all these problems into compile-time errors, operations on `fw_uint` are very strongly typed:

- All constructors are explicit: no implicit conversion from int or bool, nor from instances with a different width.
- All binary operators except multiplication require equally sized types.
- No implicit conversions to integers, bools, or instances of a different width.

These extremely strict rules turned many logic errors into compile-time errors, and made it much easier to develop robust variable precision operators.

C. Conversion Process

Each operator in FloPoCo is a C++ class which takes type specifications as constructor parameters. When an instance of this class is created, the constructor will execute C++ code which writes VHDL declarations and statements to a stream. Once the constructor has finished, the stream contains the entity and declaration for the desired FP operator. During execution the constructor may create other sub-operators, which represent sub-components such as shifters or integer multipliers. The exact statements and sub-components emitted will vary according to the class constructor parameters, and can be based on the execution of arbitrarily complex logic. Fortunately, the basic mathematical operators only rely on integer and logical calculations, with a small set of fixed-point sub-components such as multipliers and shifters.

Conversion to a templatised soft FP operator is currently a very manual process, which requires going through line by line and converting dynamic VHDL generation statements to static parameterised data-paths. For example, a typical VHDL emission statement in FloPoCo is:

```
vhdl << decl("excExpFracX",2+wE+wF)
     << "␣<=␣X"<<range(wE+wF+2, wE+wF+1)
                 << "␣&␣X"<<range(wE+wF-1, 0)<<";"<<endl;
```

which is intended to produce a pair of VHDL signal declaration and an assignment statement. For the single precisision parameters (wE=8,wF=23) it produces the following VHDL:

```
-- In architecture declarations
signal excExpFracX : std_logic_vector(32 downto 0);
-- In architecture statements
excExpFracX <= X(33 downto 32) & X(30 downto 0);
```

The FloPoCo infrastructure also takes care of automatically inserting pipeline registers, based on it's estimate of current critical path, but we can ignore this aspect as we will be relying on the HLS synthesis tool for scheduling.

To convert this statement to FloPoCo we need to convert the generation logic, *not* the VHDL that comes out of it. Most FloPoCo generation statements produce Single Static Assignment (SSA) style statements, so we can convert it to a combined C++ variable definition and assignment:

```
fw_uint<2+wE+wF> excExpFracX = concat(
     get_bits<wE+wF+2, wE+wF+1>(X) ,
     get_bits<wE+wF-1, 0>(X)
);
```

Because the statements use an SSA style we could have declared the variable as `auto excExpFracX`, but explicitly specifying the type ensures that if there is any width mis-match we will immediately get a compile-time error.

Not all widths and constants are directly based on the input type parameters, and some may be calculated during generation and stored in `int` variables in the FloPoCo code. Any such calculations must be now be calculated at compile-time, and stored as `static const int` variables, as otherwise they cannot be used for compile-time sizing of `fw_int` instances. Most such calculations are simple arithmetic expressions, but occasionally more complicated calculations such as recursive functions must be converted to template meta-programs or `constexpr` functions.

One problem with the strong type-safe approach to arithmetic comes when there are alternate branches in the datapath, and only one should be activated based on the type parameters. For example the relative widths of the input and output types changes the type of rounding performed, leading to the following code in FloPoCo:

```
if(wE>sRightSh) {
  vhdl<<decl("shVal",sRightSh)<<"␣<=␣[EXPR1]" << endl;
}
else{ //wE<=sRightSh) {
  vhdl<<decl("shVal", sRightSh)<<"␣<=␣[EXPR2]" << endl ;
}
```

where `[EXPR1]` and `[EXPR2]` represent two different complex expressions. At VHDL generation time FloPoCo would emit exactly one of the two expressions, and the other would never be emitted. Using if statements is fine in the soft core, particularly as we know that one branch will be optimised away, so we could naively convert this to C++:

```
fw_uint<sRightSh> shVal;
if(wE>sizeRightShift) {
  shVal = [EXPR1];
}else{ // (wE<=sizeRightShift) {
  shVal = [EXPR2];
}
```

However, this will usually fail to compile, as `[EXPR1]` will be malformed when `[EXPR2]` is active, and vice-versa. Even though the optimiser will completely remove the in-active code-path, it must still type-check at the C++ level.

To retain the benefit of strong type-checking while allowing for width-dependent code-paths, we include an escape-hatch called `checked_cast<W>(.)`, which can cast any `fw_uint<Wo>` to a `fw_uint<W>`, even if `Wo!=W`. This allows compilation to proceed, but it will assert during testing if the path is ever taken when the widths do not match. As long as *all* if conditions are statically determined at compile-time this is guaranteed to detect any mis-matches, though the user could accidentally add a dynamic if. This is a slight hole in the strong type-checks, but is the best that can be done until HLS front-ends support the `if constexpr` construct added in C++17.

Some FP operators use sub-components such as `LZOCShifter`, which counts leading-zeros and shifts at the same time and is used during normalisation. The operator uses $O(\log n)$ stages to implement the shifter, looking at $n/2$ bits, then $n/4$, $n/8$, ... until it reaches one bit. This maps naturally to a recursive or iterative function in C++, which is how it is implemented in the FloPoCo generator. However, in the soft implementation this will not compile: the iterative version requires the types of variables to change on each iteration, while a standard recursive version looks like infinite recursion to the compiler as it cannot determine the base case. Such components must be mapped to templatised classes rather than functions, with partial specialisation used to ensure that there is a distinct base-case at compile-time.

V. Evaluation

We will now evaluate the templatised soft FP operators, in order to determine whether they offer the claimed benefits, and also to measure some of the cost. All experiments were performed using Xilinx Vivado 2018.3, with C synthesis via Vivado HLS using default project settings (except where noted), and placement and routing using default Vivado project settings. The target device was arbitrarily selected to be a Virtex-7 7vx330tffg1157-3, though there is nothing special about this target: the operators work in other architectures, and there is no known advantage to targeting Virtex-7 rather than other parts. Unless otherwise specified the target constraint is 200MHz and all designs meet timing post place-and-rout. All resource counts and clock-rates mentioned are post place-and-route.

The focus of this evaluation is on pipelined FP performance, and specifically on the area-latency-precision tradeoffs. While the operators work perfectly well in iterative calculations, a fair

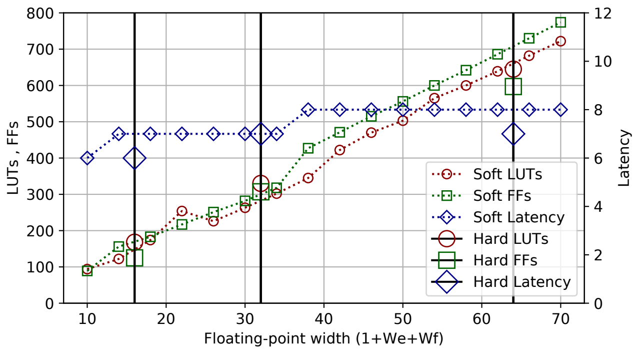

Fig. 2. Resources and pipeline latency of soft and vendor adders for increasing bit-width.

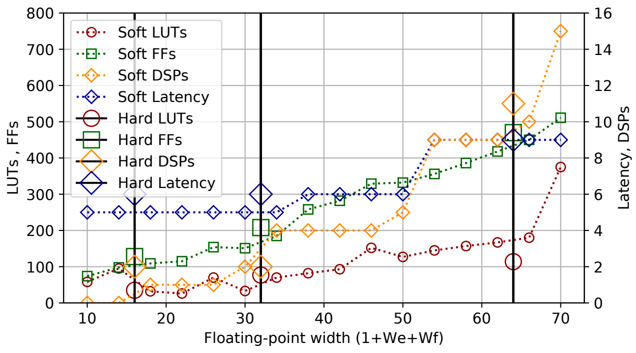

Fig. 3. Resources and pipeline latency of soft and vendor multipliers for increasing bit-width.

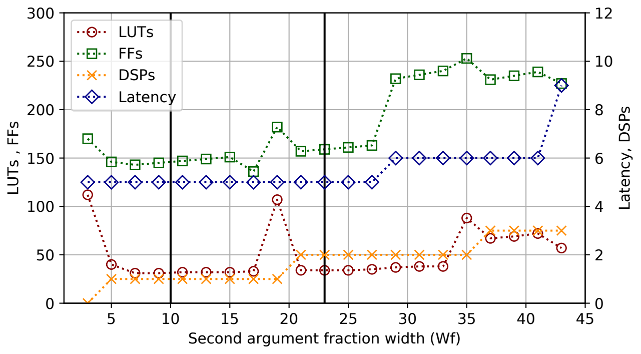

Fig. 4. Resources and pipeline latency of soft multipliers, with one argument in single-precision, and the other varying.

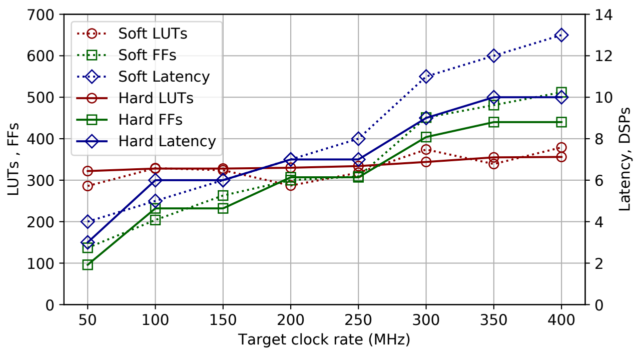

Fig. 5. Resources and pipeline latency of a single-precision adder as the target clock constraint is swept from 50 to 400 MHz.

evaluation requires larger applications, and the focus here is on the operators as primitives. As a consequence, all results here use pipelined IP interfaces using pragmas to ensure registered interfaces and an initiation interval of 1. The vendor IP blocks use variable numbers of DSP blocks: for the soft operators we rely on the synthesis tool to decide when to use DSP blocks, so where possible we try to choose the vendor variant with the same number of DSP blocks to make resource comparison more meaningful.

Our first claim is that we can support a wide spectrum of widths, which is supported by Figures 2 and 3, which shows the resource-utilisation and latency for the add and mul soft operators for a set of bit-widths between A and B, using a target clock rate of 200MHz. The exponent to fraction ratio was chosen to move smoothly through the standard precisions, and we see a very linear relationship between adder resources and width; the multiplier curves are not so smooth due to the interplay between LUTs and DSPs in the internal fixed-point multiplier. We also include the results for the vendor types at half, single, and double, and can see that the soft operators provide essentially the same area and latency as the vendor operators.

Another claimed benefit of templatised operators is that heterogeneous input types allow improved efficiency, which we support using Figure 4. This shows the impact of non-equal size inputs to multipliers, where one input is single-precision, and the other has an 8-bit exponent but a varying input size. It is worth following the DSP line, starting from 0 for a 24x4 FP multiplier, and increasing up to 3 DSPs for 24x43. At certain points the HLS tool decides to increase the DSP utilisation in the fixed-point multiplier; just before then, we see a spike in LUTs as the tool uses logic to pad the DSP multiplier. Clearly if we knows how precise each input argument actually is, then resources can be saved here.

Another scheduling benefit we see is that a single operator description can be automatically pipelined by the HLS scheduler in order to meet the current timing constraints. Figure 5 shows the variation in latency and resources as the target clock constraint is swept, showing that the single soft operator is able to track the depth of the pre-characterised IP blocks. As expected, the number of LUTs stays relatively constant, but both latency and FF count rise as the HLS tool increases pipelining. The vendor core again behaves very similarly to the soft core, although at higher clock rates the pessimism of the scheduler means that latency is a bit higher.

Once we move beyond single operators, we expect to see inter-operator merging and fusing, allowing the schedule of one operator to overlap the preceding and following oper-

TABLE I
PERFORMANCE RESULTS FOR 4 ELEMENT DOT PRODUCT WITH HETEROGENEOUS TYPES.

Parameters					Results			
Variant	TA	TL	TR	Round	LUT	FF	DSP	Delay
Soft	half	half	half	round	520	642	4	13
	single	half	half	full	1428	1255	4	12
	single	half	half	round	722	943	4	14
	single	single	half	round	900	1165	4	14
	single	single	single	round	912	1241	8	14
Vendor	half	half	half	round	655	770	8	14
	single	half	half	round	1195	1459	8	20
	single	single	half	round	1404	1672	8	19
	single	single	single	round	1313	1577	8	16

ator. We also expect to see relatively cheap full precision multiplication results, due to the ability to produce the full product. To explore this we created a parametrisable design that produces the dot product of two four-element vectors, using four multipliers at the leaves, and a tree of three adders. The design allows us to vary the type of the accumulator and the element types of the two vectors, and also supports the ability to customise the multiply operation in order to choose between “standard” (round to input type) or “full” (maintain product output precision) outputs.

Table I presents the dot product results as we vary the three types, showing that the soft approach can take advantage of scheduling and heterogeneous types. In all cases the soft version provides a lower latency than the vendor version. This is partially due to better intra-cycle scheduling of the sub-operators and some over-lapping of the operations, but also due to the FloPoCo format. The vendor cores need more steps to detect special numbers at the start of each operations, and then to convert back into IEEE format at the end of each operation – though note that because they do not support denormals, both the soft and vendor operators are doing the same calculation.

The “full” line demonstrates an interesting point, as for genuinely half-precision inputs it is providing the same precision as the completely 32-bit versions, as there is no rounding after the multiplication. An advantage is that is only requires 4 DSPs, rather than 8 for the 32-bit version, and surprisingly it is the lowest latency version. However, currently the synthesis tool seems to require a lot more LUTs and FFs to counter-balance this, so it is not an outright win. In the case of one 32-bit and one 16-bit argument we see again the advantage demonstrated earlier, as the soft version is smaller than the full 32-bit version. In contrast the vendor mixed input precision is actually bigger and higher latency, due to the need to convert the 16-bit number to 32-bit first.

An overview of the three main operators is given in Table II; other operators such as comparisons are implemented, but are less resource intensive. While the addition and multiplication operators are very competitive, the division operator is clearly much less efficient than the vendor version. Currently the division is based on an iterative division method from FloPoCo, which means that it builds a custom SRT divider in C++. This does not work well, and it would make sense to either try to use a built-in integer divider primitive from Vivado HLS, or to use a different method such as Newton-Raphson iteration [18], which is likely to map better via the HLS route.

TABLE II
COMPARISON OF SOFT (S) VERSUS VENDOR (V) OPERATORS WITH A 200MHZ CLOCK CONSTRAINT. “SYNTHESIS” SHOWS THE ESTIMATED LUT+FF COUNTS FROM HLS, WHILE “POST-PAR” GIVES THE ACTUAL RESOURCES AND ACHIEVED CLOCK RATES.

Configuration				Synthesis			Post P&R				
Op	E	F	I	D	LUT	FF	LUT	FF	DSP	Clk	Lat
add	5	10	V	7	119	165	104	147	2	229	30.5
			S	7	583	512	160	179	0	369	19.0
	8	23	V	7	411	328	330	307	0	260	26.9
			S	7	866	631	287	300	0	281	24.9
	11	52	V	7	760	650	645	597	0	224	31.2
			S	8	1502	1145	707	708	0	263	30.4
mul	5	10	V	6	39	146	35	128	2	244	24.6
			S	5	263	125	29	99	1	329	15.2
	8	23	V	6	122	240	77	208	2	233	25.7
			S	5	356	234	34	159	2	246	20.3
	11	52	V	9	199	533	114	470	11	246	36.6
			S	9	724	1174	194	495	9	331	27.2
div	5	10	N	9	224	228	219	210	0	238	37.8
			S	18	2997	1710	307	612	0	352	51.1
	8	23	V	14	805	650	757	628	0	237	59.0
			S	27	6353	3462	1017	1985	0	324	83.4
	11	52	V	33	3252	3373	3192	3338	0	239	137.9
			S	54	20545	9461	4421	9529	0	279	193.3

VI. CONCLUSION

This paper has presented a route to custom-precision FP in HLS tools via templatised soft FP cores, which allow programmers to use arbitrary FP types in their code. However, the idea provides more than just the classic approach of providing operators at given precisions, and instead allows us to specify the FP type on the arguments and return types of each operator. We have shown that this allows for area savings when input arguments are of different sizes, and also that it allows for extra accuracy to be preserved at little hardware cost. The library used to create these results is open source and available at https://github.com/template-hls/template-hls-float. It is platform agnostic, and should work in any HLS tool with a C++11 compatible front-end.

While the multiplier and adder work well, there is still substantial scope for optimisation, particularly for the divider. However, the more interesting future work involves exploiting the intelligence that can be added to the library: for example, generating constant-coefficient FP multipliers, or operators that allow fixed and FP expressions to be mixed. There is also substantial scope for restricted range operators in order to improve efficiency, for example defining types for non-negative integers allows adders to be made much smaller. Given a richer set of types, the most efficient operator could be built based on knowledge about what possible values it could take, for example allowing NaN handling logic to be removed.

References

[1] IEEE Standards Committee, "IEEE standard for floating-point arithmetic," pp. 1–70, Aug 2008.

[2] I. Buck, "Taking the plunge into GPU computing," in *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley Professional, 2005, ch. 32.

[3] N. Higham, "Half precision arithmetic: fp16 versus bfloat16," https://nickhigham.wordpress.com/2018/12/03/half-precision-arithmetic-fp16-versus-bfloat16/, 2018.

[4] *BFLOAT16 – Hardware Numerics Definition*, https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, 2018.

[5] M. Langhammer and T. VanCourt, "Fpga floating point datapath compiler," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 259–262.

[6] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA '08, 2008, pp. 260–260. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344717

[7] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang, "Automating customisation of floating-point designs," in *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*, ser. FPL '02, 2002, pp. 523–533.

[8] Xilinx, "Pg060: Floating-point operator v7.1 - logicore ip product guide," https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf, 2017.

[9] Intel, *Floating-Point IP Cores User Guide*, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altfp_mfug.pdf, 2016.

[10] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[11] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1–20:17, Jul. 2016.

[12] J. Detrey and F. de Dinechin, "Fplibrary, a vhdl library of parametrisable floating-point and lns operators for fpga," 2004.

[13] J. Allan and W. Luk, "Parameterised floating-point arithmetic on FPGAs," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing.*, vol. 2, 2001, pp. 897–900.

[14] "Pipelined floating-point library manual," Celoxica Ltd., 2005.

[15] J. Hauser, "Berkeley softfloat," http://www.jhauser.us/arithmetic/SoftFloat.html, 2018.

[16] 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 Conference Exhibition (DATE)*, March 2018, pp. 37–42.

[17] Y. Uguen, F. de Dinechin, and S. Derrien, "Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8.

[18] B. Pasca, "Correctly rounded floating-point division for dsp-enabled fpgas," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 249–254.