Generic Arithmetic Units for High-Performance FPGA Designs

John R. Humphrey, Daniel K. Price, Michael R. Bodnar, Petersen F. Curt, James P. Durbano
Accelerated Computing Division
EM Photonics, Inc.
51 E. Main St. Suite 203, Newark, DE, 19711
USA

Abstract: As PLD applications expand to include high-speed computing, the number of data types required grows commensurately. Current applications of programmable devices require operations on various integer, fixed-point, and floating-point precisions, in both real and complex (a+jb) formats. Moreover, simulating large, high-precision systems that include arithmetic units requires significant time. In this paper, we present a method of VHDL programming that addresses the growing number of data types and simulation time to allow for rapid development and simulation of mathematical-based hardware designs.

Key-Words: Floating-point, arithmetic, precision, complex, generic, framework, VHDL, FDTD

1 Introduction

As PLD applications expand to include high-speed computing, the number of data types required grows commensurately. Current applications of programmable devices require operations on various integer, fixed-point, and floating-point precisions, in both real and complex (a+jb) formats. Moreover, simulating large, high-precision systems costs a burdensome amount of time due to simulation of the logic of the design’s arithmetic units. In this paper, we present a method of programming in VHDL that addresses the issues that arise with expanding number of data types and simulation times.

To implement this method, we built an arithmetic infrastructure that models the concept of “generic programming.” Generic programming is found in high-level software languages, such as C++, and allows the programmer to express ideas in a format independent of data types; common uses for this are data containers and searching/sorting algorithms [1]. While VHDL does not explicitly support this sort of programming (it is contrary to the device-level nature of the language), we implemented this idea through a novel use of existing parameterization and subprogram features of the language.

We developed a suite of VHDL units that can perform operations on various data types: 32- and 64-bit floating-point values in both real and complex formats. The operations we developed included addition, subtraction, multiplication, accumulation [2], storage, and time-delay. These flexible units enabled the quick, efficient development of more complex designs, such as matrix-vector multiply, without focusing on the underlying representation of the arithmetic units and their data type at the algorithmic level. Additionally, these arithmetic units can utilize VHDL’s built-in “real” type [3] to simulate complex
systems drastically faster, by changing a single constant at the top level of the design. Furthermore, the synthesis and simulation overhead associated with these generic units is negligible, making this system quite powerful for developers. We will first describe the unique problems that are encountered when solving this problem, and then go on to describe how our solution solved those problems. We will then discuss the effectiveness of our solution by describing several cases studies that used our approach.

2 Motivation
As described above, a designer often wants to incorporate several floating-point formats in a design, or maintain several configurations of a design that utilize different formats. One way to accomplish this is to develop different implementations of a base design. Each version would consist of the same basic logic blocks, except for different arithmetic units. This approach is susceptible to errors and is difficult to maintain and scale for many reasons. For example, any fundamental system update must be replicated multiple times across each of the implementations. Motivated by the disadvantages of this approach, we set out to develop a framework whereby multiple data types could be supported via a single architecture.

Three key challenges are encountered when designing a unified framework that solves the above goals: the arithmetic complexity of a particular operation, the differing latencies between different formats of the same operation, and lengths simulation times required by large systems.

2.1 Arithmetic Complexity
Different arithmetic operations for different formats have different computational complexity. Complex arithmetic requires more computational units than real arithmetic. For example, a real addition, or multiplication, requires one adder, or one multiplier, respectively. A complex addition, on the other hand, requires two adders: one for the real portion, and one for the imaginary portion. For complex multiplication, even more units are required: four multipliers and two adders. A goal of generic programming is that a designer should not have to know how an underlying calculation takes place [1], and thus we need to provide a framework that provides this abstraction.

2.2 Latency
In addition to requiring different number of computational units, each operation has a different latency. This results from the complexity of a particular operation; for example a double-precision addition requires more pipeline stages than its single precision variant. Complex multiplication further complicates matters, as it requires that adders be chained to the output of the multipliers, yielding a combined delay that is roughly twice that of the real multiplications. A developer should not need to design for these inconsistent latencies, and so the framework should provide a mechanism for automatic parameterization for different operations.

2.3 Simulation Time
Simulation of large systems takes a burdensome amount of time, especially when floating-point arithmetic is involved. A significant portion of this time is
due to simulation of arithmetic units. This is often unnecessary, as these units typically have been verified independently. A framework that reduces the time spent simulating would allow for the implementation of new features, and an increase in productivity.

3 Implementation Details
To solve the problems presented above, we designed a generic floating-point signal and framework of generic floating-point arithmetic units that utilized the new signal. In this section we discuss how we implemented this solution, how it solves the above problems, and the advantages that it provides.

3.1 Developing a Floating-point Signal
The first implementation step was to develop a “generic” floating-point signal, called “FP”. This FP type was defined in VHDL as a record consisting of 6 components (Table 1).

<table>
<thead>
<tr>
<th>Name</th>
<th>Underlying Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r32</td>
<td>32-bit STD_LOGIC_VECTOR</td>
<td>Single-precision, real</td>
</tr>
<tr>
<td>c32</td>
<td>32-bit STD_LOGIC_VECTOR</td>
<td>Single-precision, complex</td>
</tr>
<tr>
<td>r64</td>
<td>64-bit STD_LOGIC_VECTOR</td>
<td>Double-precision, real</td>
</tr>
<tr>
<td>c64</td>
<td>64-bit STD_LOGIC_VECTOR</td>
<td>Double-precision, complex</td>
</tr>
<tr>
<td>rf</td>
<td>Simulator’s built-in FP type</td>
<td>Real</td>
</tr>
<tr>
<td>cf</td>
<td>Simulator’s built-in FP type</td>
<td>Complex</td>
</tr>
</tbody>
</table>

This signal incorporates all combinations of single/double precision and real/complex floating-point representations. Additionally, it contains the simulator’s non-synthesizable built-in floating-point type, the advantages of which will be described later. In order to accomplish common floating-point operations on this signal, such as sign retrieval, negation, and bundling/unbundling of FP signals, we built support functions and procedures. These functions and procedures eased the task of working with the FP signal, and aided in generic programming by packaging common functionality.

It should be noted that for this work we limited the data types to the canonical floating-point variants (real and complex, 32- and 64-bit). This work could easily be extended to use fixed-point, integer, or even nonstandard floating-point sizes.

3.2 Building an FP Type Framework
The FP signal, by itself, is of little value without a framework of units that can operate on this signal. Before designing this framework, we had four major arithmetic units: adders and multipliers in both single- and double-precision formats. These units accepted the STD_LOGIC_VECTOR type, and their correct operation had been verified many times. To avoid modifying these units, and potentially introducing errors, we designed a wrapper for each arithmetic unit that would handle the task of accepting the new FP signal.

Each wrapper is responsible for instantiating the specific components needed for a particular type of computation, as described in Section 2.1. In addition to accepting the FP signal, the
wrapper accepts a generic type that indicates which floating-point format to instantiate. To a designer, the wrapper is the arithmetic unit, as he or she simply instantiates the wrapper whenever a particular operation is desired, and passes along the generic to denote what type of floating-point computation to perform.

![Diagram of Arithmetic Wrapper](image)

**Fig. 1 – Designing with arithmetic wrappers**

Here we see the arithmetic wrapper instantiating the required low-level units for a floating-point format indicated by the generic. The inputs to the wrapper are the operands and generic, and its output is the result in the indicated format.

As shown in Fig. 1, a designer need not know which components the wrapper unit instantiates for a particular floating-point format. This solves the problem that is presented in Section 2.1, in that it provides a unified interface, and abstracts the computational complexity from a designer, satisfying the goal of generic programming. Additionally, this generic can be chained through a system to provide all units with the same format. In this way, the generic need only be changed at the top-level to change the format of all arithmetic units in a system.

For a brief motivating example, consider a matrix-vector multiply (MVM) unit. Fundamentally, this is a multiplication of two numbers followed by an accumulation. There may also be some memory required for storage of intermediate results. In such a case, a user of this framework would define a unit that accepts the FP type and the generic describing the type of that data. Then, the user instantiates wrapper FP units for the multiplication, accumulation, and storage without knowledge of the data type – the generic is simply passed to the wrapper units with the FP data. The end result is an MVM unit that is applicable to all data types the generic wrapper units can handle.

As described in Section 2.2, however, a problem still remains in that pipeline lengths are not the same for all types of computations for a particular unit. To incorporate these differences in pipeline lengths, a function is provided for each wrapper that provides information for the parameterization of the design.

This function’s input is the same generic as is passed to the arithmetic unit, and its output is the latency of the particular format in question. With this information, a designer can instantiate a floating-point delay unit to hold the FP signal as needed to match pipeline latencies with other units.
Fig. 2 – Matching Latencies

Here we see an arithmetic unit matched to the latency of some other unit. This is accomplished via a calculable variable delay.

Because the unit to match with has a fixed latency, and the arithmetic wrapper has a known latency (by way of the latency function), one can calculate the delay required to match latencies between units. This method solves the latency mismatch problem presented in Section 2.2.

3.3 Advantages

The FP signal type and associated framework has considerable advantages over our previous implementation of floating-point arithmetic. Most importantly, this approach provides us with a unified design for single/double-precision and real/complex types. For example, if a system were to exclusively use double-precision real arithmetic, the FP type at the uppermost level need only be changed, as mentioned in above.

Another advantage of this approach is that it abstracts the details of floating-point computation from the design hierarchy, as described in Section 3.1. A designer need not know how a particular computational format is implemented to use it. This allows for high-level, generic-programming-like design of a system with minimal focus on details. A high-level design approach is also aided by the built-in simulation type that provides high-level design simulation.

As was shown in Table 1, the new FP type has built-in constructs for design simulation. If the rf or cf type is specified, the arithmetic units will utilize the simulator’s built-in floating-point type to perform simulation of the system.

This is an incredibly important optimization that solves the problem presented in Section 2.3, as it saves the HDL simulator from directly simulating our custom floating-point units. Because our arithmetic units have been proven many times over, we need not require the simulator to validate them for each system simulation. This greatly decreases the amount of time required for simulation, thus enabling rapid prototyping and verification of new system features.

4 Conclusion

As hardware-based accelerated computing grows, it is vital that the tools and techniques used to develop the accelerated architectures grow accordingly. In this paper, we presented a “generic” floating-point infrastructure. Units that utilize this infrastructure are capable of operating on a variety of data types, do not increase the size of the design appreciably, and greatly simplify the design process. We used this approach in the design of our FDTD system, and found it to maintain the same accuracy and correctness as the previous implementations, while providing minimal overhead. Additionally, it greatly decreased the amount of time required for system simulation, and allowed us to focus on other features. This framework will greatly expedite the development of future FPGA designs, thus enabling the rapid simulation and deployment of hardware-accelerated projects.

In future work, we will analyze the benefits and limitations of this approach. This will include analysis of accuracy, speed tradeoffs, and simulation time tradeoffs for multiple hardware systems.
References:

