A Fixed Point VHDL Component Library for a High Efficiency Reconfigurable Radio Design Methodology

Scott D. Hoy, Marco A. Figueiredo

Abstract— Advances in Field Programmable Gate Array (FPGA) technologies enable the implementation of reconfigurable radio systems for both ground and space applications. The development of such systems challenges the current design paradigms and requires more robust design techniques to meet the increased system complexity. Among these techniques is the development of component libraries to reduce design cycle time and to improve design verification, consequently increasing the overall efficiency of the project development process while increasing design success rates and reducing engineering costs.

This paper describes the reconfigurable radio component library developed at the Software Defined Radio Applications Research Center (SARC) at Goddard Space Flight Center (GSFC) Microwave and Communications Branch (Code 567). The library is a set of fixed-point VHDL components that link the Digital Signal Processing (DSP) simulation environment with the FPGA design tools. This provides a direct synthesis path based on the latest developments of the VHDL tools as proposed by the IEEE VHDL 200x, which allows for the simulation and synthesis of fixed-point math operations while maintaining bit and cycle accuracy. The VHDL Fixed Point Reconfigurable Radio Component library does not require the use of the FPGA vendor specific automatic component generators and provide a generic path from high level DSP simulations implemented in Mathworks Simulink to any FPGA device. The access to the component synthesizable, source code provides full design verification capability.

Index Terms— Software Defined Radio (SDR), Reconfigurable Computing, Very-High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), Field Programmable Gate Arrays (FPGA), Digital Signal Processing (DSP).

I. INTRODUCTION

A Generic component library has been created to facilitate the development of Software Defined Radios (SDR), here referred to as Reconfigurable Radios to represent the fact that the components are designed for an FPGA-based reconfigurable computing platform. A new computer design paradigm has been established with the advent of FPGAs. As a consequence, a more robust design methodology is required to address the increase in system complexity. A new design methodology is described in this paper, along with the definition of the VHDL Fixed-Point Reconfigurable Radio Component Library.

The new emerging computer design paradigm is better abstracted when the traditional Von-Neumann compute model is viewed as a subset of the new paradigm, in contrast with the conventional view that FPGAs are a subset of the traditional design paradigm. In this more advanced view, the algorithm in question is implemented first in a simulation environment and later goes through a process of synthesis to be implemented in an FPGA centric environment that encompasses one or more Von-Neumann processors.

The rationale for the emergence of a new computer design paradigm is justified in the first section of the paper. The second section of the paper describes the proposed design methodology that addresses the more advanced view of computer systems design. The next section describes the generic component library, followed by a discussion of the issues addressed in the design of the library and the advantage of the approach when compared to technology-dependent component generators.

The uniqueness of the VHDL Fixed-point Reconfigurable Radio Component Library is that it uses the new proposed IEEE VHDL fixed-point package that the IEEE will release in the next VHDL standard, currently called VHDL-200x.

Appendix A of the paper discusses the science of paradigm shifts and appendix B presents the fixed-point arithmetic notation used in the design of the component library. A slide presentation that describes a design example utilizing the component library complements this paper [1]. The example is a M\textsubscript{ary} Phase Shift Keying (M-PSK) Quadrature Amplitude Modulator (QAM) that was developed in a period of five days.
II. THE PARADIGM SHIFT IN COMPUTER DESIGN

As a paradigm nears its point of exhaustion, a new one, much more powerful and complex, starts its evolution. The new emergent cycle is covered by the current and successful paradigm, and clouded by its own state of disorder. As the new paradigm is consolidated, it suddenly appears from nowhere. There is enough evidence that shows the existence of a paradigm shift in computer design. For about half a century, the software paradigm, based on the Von Neumann compute model, has been the major force in the development of computer technology. The advent of commercial Field Programmable Gate Arrays in 1985 started a new cycle with the exploration of reconfigurable computing techniques [16]. Today, reconfigurable computing technology enables the emergence of a new computing paradigm that is orders of magnitude more complex and powerful than the software paradigm.

Reconfigurable computing utilizes the hardware paradigm to build applications. However, a reconfigurable FPGA allows a new configuration, or hardware design, to define its logic circuit in a fraction of time. As a result, a reconfigurable computing engine allows multiple hardware designs to be time multiplexed and enables several applications to share the same device, similar to what happens in the software paradigm defined by the Von Neumann compute engine, but offering processing speeds similar to that of ASICs. As a result, a new, more powerful, and more complex paradigm is born, the reconconfigurable computing paradigm.

Those ignoring the computing paradigm shift currently under way prefer to see FPGA devices as just another element within the current computer design paradigm, where the Von-Neumann Central Processing Unit (CPU) is the heart of the system and every other device is a peripheral to it. This current design paradigm has created classes of design engineers separated as CPU designers, software designers and hardware designers. Another possible classification is that of an embedded systems designer to describe one that can both integrate devices and program them.

Following the current classifications, the roles of each class of designer is very well determined and there is very limited crossing between the boundaries of one professional and another. For example, a CPU designer specializes in mathematical logic units such as floating-point arithmetic logic units (ALU), and complex memory management units and cache memory designs. It is very seldom that a traditional hardware or embedded systems designer has to design mathematical functions of high logic complexity that a CPU designer needs to implement. Conversely, software engineers use high level language compilers to map algorithms to a Von-Neumann processor and its operating systems, but they do not have to understand the concept of signals switching within the boundaries of a periodic clock, a domain better know by traditional hardware designers.
In the traditional computer design paradigm, FPGAs are just devices to be embedded in the platforms to allow for a more flexible reconfiguration of the system. However, if one looks closer into the current needs of a hardware designer while implementing an FPGA application, one realizes that the traditional professional borders are being crossed and the level of design complexity is growing for the average designer. Traditional hardware designers creating FPGA applications need to understand issues that are normally in the realm of CPU designers, such as the handling of complex arithmetic operations, either in floating or fixed-point representations. Once in the hands of a few, highly specialized designers, these functions now need to be implemented by the average FPGA designer. This is a clear indication that the traditional computing design paradigm is being broken. As a result, the higher level of complexity required by the average engineer requires a more sophisticated design paradigm to emerge.

Figure 3 – Reconfigurable Computing Design Paradigm

The computer design paradigm that is emerging is better abstracted when the Von-Neumann compute model is viewed as a subset of the new paradigm. In this more advanced view illustrated in figure 3, the algorithm in question is implemented first in a simulation environment and later goes through a process of synthesis to be implemented in an FPGA centric environment that encompasses one or more Von-Neumann processors. This is the approach utilized in this work to implement reconfigurable radio designs, i.e., radio systems that are implemented utilizing Digital Signal Processing (DSP) techniques in an array of FPGA devices. The breakthrough in adopting such a higher-level view of reconfigurable systems design results in a more efficient and robust design methodology that needs to be followed in order to handle the higher level of design complexity.

III. A NEW DESIGN METHODOLOGY

In order to overcome the inherent complexities of the reconfigurable computing design paradigm and provide a path for efficient design implementations by the average engineer, a few modifications to the current design paradigm must be addressed. First, it is imperative that the application design be looked at from a multi-disciplinary point-of-view. As such, the algorithm developer, many times referred to as the application scientist, is provided with a design environment where he/she can develop a model that is platform independent. This environment allows the scientist to create and test the model while transferring the design requirements to the application engineer.

Second, from the scientific model, the application engineer should be provided with tools that allow him/her to derive the solution according to the target platform. The approach adopted in this work is to use VHDL, an industry-standard hardware description language (HDL), to which the model is synthesized to target the reconfigurable computing platform. The components that target a Von-Neumann compute engine are properly mapped using the traditional software design paradigm, as illustrated in figure 4.

Figure 4 – DSP Application Development Flow

Third, as in any complex application design, the documentation of the design process becomes paramount to its success. Starting with a requirements document that defines the system to be developed, a document that details the chosen design architecture follows. At the end, a characterization of the working design results in a final document. In short, design documentation is essential in an efficient design methodology that leads to repeatable implementations and allows teamwork.
Eventually, design automation can provide a path for design implementation directly from the requirements document.

Finally, in order to reduce complexity, generic component libraries are used to reduce the need for the average engineer to implement every component within the application domain. These libraries must be generic enough to allow the application to be target to diverse underlying technologies.

IV. THE RECONFIGURABLE RADIO COMPONENT LIBRARY

A VHDL Fixed-point Reconfigurable Radio Component Library has been developed to aid in bridging the gap between migrating advanced DSP algorithms that are developed in high level modeling tools down to a hardware description language (VHDL), so they can be realized in a real-time reconfigurable computing platform. The intent of the library is to accelerate the conversion process from the high level algorithm modeling tools to synthesizable HDL code. The algorithm designer and the FPGA application designer can use existing design tools to synthesize the design while maintaining complete control over the design architecture.

The algorithm designer implements the scientific model using floating-point mathematical representation in a simulation and modeling tool. The model is then translated to fixed-point representation and checked against the floating-point model using the simulation tool. The VHDL library is structured to allow the easy migration of the fixed-point model to a synthesizable fixed-point VHDL model that can be delivered to the FPGA designer for integration into the reconfigurable computing platform. For the scope of this paper, the high level algorithm modeling tools used are Matlab and Simulink, from MathWorks.

The diagram above shows the model based algorithm design flow used. This diagram illustrates that there is both a hardware component and software component in implementing high level algorithms into a reconfigurable computing platform. The hardware integration with supporting control software is the primary focus for the development of the VHDL fixed-point library. The supporting control software is contained in the DSP C library named dslibc. The DSP C library is a complementing software library that contains a fixed-point data type that is identical to the VHDL fixed-point library and provides supporting math functions that are used in configuring the VHDL fixed-point components, i.e., filter coefficient calculations.

The primary function of the VHDL Fixed-point Reconfigurable Radio Component Library is to capture the look-and-feel of Simulink within the current definition of the VHDL language specification. The components that are in the VHDL Fixed-point Reconfigurable Radio Component Library attempt to have a one to one mapping to its Simulink counterpart. The use of the library implies performing block substitution of the Simulink block to its equivalent VHDL fixed-point component. Therefore, the Simulink model diagram is recoded in VHDL fixed-point by connecting the equivalent VHDL fixed-point components that match the Simulink components. This approach provides complete and absolute control over the VHDL code design of a given algorithm. Nothing is hidden from the designer. The designer can freely examine all levels of the design hierarchy of the algorithm that is being designed. This approach promotes design reuse by creating highly generic synthesizable VHDL components.

The primary obstacle in translating any high level fixed-point algorithm to synthesizable VHDL is the lack of a standard fixed-point data type. A standard fixed-point VHDL data type guarantees bit accuracy between the VHDL and the high level algorithm modeling tool domains. The uniqueness of the VHDL Fixed-point Reconfigurable Radio Component Library is that it uses the newly proposed IEEE VHDL fixed-point package that the IEEE will release in the next VHDL standard, currently called VHDL-200x. The new IEEE VHDL fixed-point package adds a native fixed-point data type to VHDL. The package is written so that it is compatible with the current VHDL standard release. The package also works with current commercial VHDL simulators and synthesis tools. The reader should refer to Appendix B and/or reference [5] for additional information on fixed-point notation and the proposed VHDL fixed-point package.

The VHDL Fixed-point Reconfigurable Radio Component Library is organized into three separate libraries: the fixed-point library (fixptlib), the standard logic vector library (slvlib), and the application library (applib). The fixptlib and the slvlib contain the low level components that attempt to be as generic as possible to strongly promote design reuse. The fixptlib uses only the VHDL-200x proposed fixed-point data types (sfixed and ufixed) and the slvlib is limited to the data type std_logic_vector or std_logic.
The slvlib contains basic logic components such as flip-flops, registers and comparators; memory components such as Random Access Memory (RAM) and Read Only Memory (ROM) blocks; communication level converters (Non Return to Zero (NRZ) L, M and S); and clock divider components.

The fixptlib contains real and complex math operators such as adders, subtractors, multipliers and dividers; multi-rate components such as variable down-sampler and up-sampler, and decimating and interpolating Cascaded Integrator-Comb (CIC) filters; real and complex memory components such as unit delays, variable unit delays, sine and cosine memory units; real and complex filter components such as Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters; communication components such as Numerically Controlled Oscillators (NCO), I/Q vector modulators, second order Digital Phase Locked Loop (DPLL) filters, bi-phase coders, symbol synchronizers, Costas loop, and pseudo-random number generators; and control logic components such as comparators and counters.

The library applib contains complete systems or applications that are built out of the components that are in the fixptlib and slvlib. This growing library is currently composed of a multi-rate Binary Phase Shift Keying (BPSK) demodulator, a multi-rate Quadrature Phase Shift Keying (QPSK) demodulator, a NASA ground network demodulator and a Tracking Data and Relay Satellite System (TDRSS) Direct Sequence Spread Spectrum (DSSS) de-spreader with a multi-rate demodulator, which includes all the TDRSS spreading codes.

The applib library is not required to be generic since the components in the library define the highest levels of hierarchy for a complete system to be integrated into the FPGA implementation design flow. The components that are in the applib convert any fixed-point components that use the VHDL-200x fixed-point data types to the std_logic_vector data type by stripping off the binary point. The integer value is preserved. All components in the applib use the current VHDL standard data type std_logic_vector or std_logic on their inputs and outputs.

V. LIBRARY DEVELOPMENT

One major issue that the designer faces in translating a fixed-point Simulink model to VHDL fixed-point is pipelining. Pipelining a fixed-point algorithm in VHDL can be just as tedious as maintaining the bit accuracy. In Simulink, pipelining is not typically done since the algorithm design is the main focus. If feedback is used in an algorithm design such as a Phase-Lock-Loop (PLL), equalizer or demodulator, the pipeline delay can be modeled in Simulink by placing a single variable delay on the feedback. This delay value can be the total pipeline delay that will be in the VHDL fixed-point model to maintain bit accuracy or can be adjusted to determine the maximum amount of pipeline delay that the system supports. The VHDL model distributes this delay throughout its architecture for speed. All components in the library contain pipelining variables in their generic maps to simplify the pipelining process. This attempts to make pipelining a simple design change versus manual recoding of existing VHDL, if pipeline registers are needed to be moved or re-arranged. The use of the fixed-point data type coupled with the pipeline control now allows for simplified modifications to the fixed-point bit precision or pipeline registers in the VHDL code.

Commercial HDL code generators that generate HDL code from Simulink have been examined, but due to their current limitations they are not used in the critical design path on large scale designs. They do work well for small, simple designs, but they begin to impose additional limits and work on a designer when creating large scale multi-hierarchical DSP systems. An example would be a multi-rate DSP algorithm such as a demodulator. Additional issues include not supporting native data types, toolboxes, or block sets that are available in the Matlab and Simulink tools. Examples are multi-dimensional arrays, fixed-point data type, stateflow, simulation accelerator, and the validation and verification tools.

The current HDL code generators are implemented by third party vendors to support their underlying FPGA technology, which requires the application designer to use of their block sets to handle the code generation. This can lead to having to re-draw a fixed-point Simulink model that is done using the Mathworks blocks in the third-party code generator vendor blocks. This approach creates additional work that the user has to perform when using the current code generators, an issue commonly referred to as "block set hell". When the designer attempts to generate code, the designer has very limited to no control over how the code is generated. Most likely the generated code is usually not intended to be viewed outside the code generation process but treated as a "black box", which can lead to verification limitations. A critical question to understand on the code generation tools is how Simulink time is translated to VHDL time and the corresponding HDL clock tree that will drive the hardware logic. This concept can be difficult to grasp for both the algorithm designer and the FPGA designer.
and the C++ templates. The improved generics allow for generic arrays such that the element width along with the dimension can be defined at instantiation. The current VHDL standard only allows for the dimension to be defined at instantiation but with the element width fixed. The new standard, for example, allows the creation of a generic vector or matrix that could have each array element of type std_logic_vector but the width of the std_logic_vector can be defined at signal or variable instantiation, along with the vector/matrix dimensions. New data types that significantly aide in higher level abstractions is the addition of a synthesizable fixed-point and floating point data type along with supporting synthesizable math algorithms such as square-root, sine, cosine, etc. The addition of these data types significantly aide the designer in the migration of a fixed-point design that was originally modeled in a high level tool, such as Simulink, to VHDL since it makes maintaining bit accuracy trivial.

It is important to keep in mind the design limitations that currently exist between the VHDL language standard and the high level algorithm modeling tools such as Matlab/Simulink. VHDL currently limits the designer by not allowing higher levels of abstraction due to limits on generics and current lack of higher level data types. The main limitation with VHDL can be traced to the synthesis tools by asking the question: How compliant is the synthesis tool to the latest VHDL standard release? Obtaining a complete answer to this question from a synthesis tool vendor is difficult and usually countered by write the VHDL code and run it through the synthesis tool to see if the expected logic was generated without the tool complaining.

The high level algorithm modeling tools such as Matlab/Simulink have their own limitations when attempting to translate a fixed-point model to a HDL. The main issue with the translation is that Matlab/Simulink is not a traditional logic design tool and its main design entry is based on block diagram or signal flow abstraction rather than a schematic abstraction. When a design is translated from a high level algorithm modeling tool down to the HDL level, additional information is inherently added to the design at the HDL level due to moving to a lower level of abstraction. The objective for a HDL code generator is how to present the additional low level details that must be addressed in crossing tool domains (high level abstraction to low level abstraction) while preserving (as much as possible) the existing design flow of the high level algorithm modeling tools.

While VHDL and Simulink have their limitations that the designer needs to be aware of, both have some similarities that the designer can use to his/her advantage. Both VHDL and Simulink enforce a model based design approach through a strong design hierarchy, albeit Simulink is strictly visual, coupled with some scripting language (Matlab), while VHDL is text based. The addition of the fixed-point package to VHDL in VHDL 200x significantly improves translating a high level fixed-point algorithm into VHDL fixed-point by resolving the bit accuracy.

A final note concerning the third-party HDL code generators from Simulink fixed-point models is that Mathworks has entered into this arena and has recently released a Simulink to HDL code generation tool this month, September 2006, with the R2006b release. According to the release notes, it does support native Simulink blocks that support the fixed-point data type and HDL code generation from Stateflow (state machine editor). The HDL code generation from a high level fixed-point architecture block diagram model is possibly the future in improving the designer productivity, but these tools are currently in their infancy.

VI. CONCLUSION

The work performed in the development of the VHDL Fixed-point Reconfigurable Component Library took in consideration the emergence of a new computer design paradigm. Consequently, it led to the development of a more robust and efficient design methodology that addresses the higher levels of complexity present in reconfigurable computing design. By recognizing the potential of the new features presented by the upcoming IEEE VHDL standard, the component library not only reduces the time to assemble reconfigurable radio applications, but it also leverages on current Commercial Off-The Shelf (COTS) design tools, while providing a seamless integration path between the simulation and synthesis environments. The result is a considerable reduction of time in application development, from months to days or weeks, depending on the application. The example design created for the purpose of demonstration of the library capabilities, an M-PSK modulator, was implemented in five days, a process that traditionally takes weeks or months.

This work validates the conclusion that generic VHDL component libraries provide a path for the increase in design efficiency and contribute to the establishment of the new reconfigurable computing paradigm. This technique should be extended to application fields other than reconfigurable radios. Finally, the proposed improvements in the IEEE VHDL standard haven proven to be essential for the success of the new design approach.

APPENDIX A – PARADIGM SHIFTS

The evolution of a paradigm is said to pass through three distinct phases\footnote{11, 12}. In phase 1, the formation phase, the original standard is invented or formed from the combination of several elements previously disorganized. In phase 2, the consolidation phase, the new standard is perfected in a cumulative process through repetitive modifications and
rejection of the pieces that do not fit the original standard. In phase 3, the transformation phase, the original standard is broken, and differences previously left out are incorporated through innovation and opening of the system. Invention, perfection, and innovation characterize the three phases of development of a system.

The three phases of the evolution of a system can be illustrated by an S curve, as shown in figure 1. The phases are marked by two breakpoints, or points of change.

The first breakpoint marks the change from phase 1, where an accepted paradigm has been established, and phase 2, where the goal is to improve the established paradigm. It is only possible to reach higher levels of growth through repetition, amplification and perfection of the basic paradigm. However, every successful paradigm enjoying the growth in phase 2 extends its reach to a certain level where the original paradigm is exhausted, and it needs to be changed. At this second breakpoint, the ideas that have been rejected in phase 2 are revisited.

Two distinct processes characterize phase 3, the transformation phase. The differences are integrated through the means of innovation and construction following the basic paradigm that appeared in phase 2, and the old system is continuously integrated through assimilation and connection of the parts previously left out. At the same time, a new phase 1 is created, and a bifurcation appears, as illustrated in figure 1. There is a division, where part of the established paradigm moves away from the process of innovation and begins a process of reinvention of the entire paradigm in a much more complex level.

However, the new emergent cycle, or the new phase 1, is invisible, covered by the current and successful paradigm, and clouded by its own state of disorder. As it reaches its phase 2, it suddenly appears from nowhere.

**APPENDIX B – FIXED-POINT NOTATION**

Fixed-point notation is illustrated in figure 6. Fixed-point notation partitions an integer into three different parts. These parts are the sign bit for signed case (denoted by the letter s which implies two's complement), integer bits (denoted by the letter i), and fractional bits (denoted by the letter f).

Signed Fixed-Point (Two's Complement):

<table>
<thead>
<tr>
<th>s</th>
<th>i</th>
<th>i</th>
<th>i</th>
<th>i</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
<td>2^6</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
<td>2^-1</td>
<td>2^-2</td>
<td>2^-3</td>
<td>2^-4</td>
<td>2^-5</td>
</tr>
</tbody>
</table>

Binary Point

**Figure 6- Fixed-Point Notation**

Unsigned Fixed-Point:

<table>
<thead>
<tr>
<th>i</th>
<th>i</th>
<th>i</th>
<th>i</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
<td>2^6</td>
<td>2^5</td>
<td>2^4</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
<td>2^-1</td>
<td>2^-2</td>
<td>2^-3</td>
<td>2^-4</td>
</tr>
</tbody>
</table>

Binary Point

**Figure 7- Fixed-Point Variable**

The binary point separates the fractional or precision bits from the integer bits. A fixed-point variable can be described by the following three attributes: the total number of bits, wordlength (wl), total number of integer bits, integer wordlength (iwl), and the total number of fractional bits, fractional wordlength (fwl). From the three attributes, only two are needed since the third one can be derived from the two attributes that are used to define a fixed-point variable. Simulink, for example, defines a fixed-point number using the wordlength and fractional wordlength. The VHDL 200x fixed-point package defines a fixed-point variable based on an array range that contains both positive and negative indices. In VHDL, the binary point is assumed to be placed between the indices 0 and -1.

Using the fixed-point variable shown in figure 7 the VHDL signal declaration to describe this fixed-point variable is as follows:

- Signed Case: signal x : sfixed(5 downto -6);
- Unsigned Case: signal x : ufixed(5 downto -6);

The standard VHDL attribute x'length can be used to determine the wordlength. For the signed case, to determine number of integer bits the VHDL attributes x'high or x'left can be used. For the unsigned, case to determine the number of integer bits a +1 needs to be applied to the x'high or x'left due to the zero indexing. The number of fractional bits can be determined by abs(x'low) or abs(x'right). This will return a positive number for the number of fractional bits. If the absolute value function is omitted, the returned number will be negative if the number of fractional bits is greater than zero. The VHDL fixed-point package defines a to_real() function that will convert the fixed-point variable to a floating-point representation that is based on the defined fixed-point attributes. For the fixed-point
variable defined in figure 7, the signed real value is -10.40625 and the unsigned value is 53.59375.

The VHDL 200x fixed-point package provides support for fixed-point rounding and overflow. The rounding modes supported are truncation and round to nearest. These are denoted by the defined boolean constants fixed_truncate and fixed_round. The overflow modes supported are saturation and wrap. These are denoted by the defined boolean constants fixed_saturate and fixed_wrap.

The VHDL 200x fixed-point package defines the fundamental fixed-point arithmetic operators addition (+), subtraction (-), multiplication (*), division (/), modulus (mod), remainder (rem), absolute value (abs), and reciprocal. The fixed-point arithmetic performs full precision arithmetic. This means that the user must be aware of what the resulting fixed-point variable wordlength is for the fixed-point arithmetic that is performed so that the variable that will contain the result will be the proper length. Table 1 illustrates the full precision fixed-point arithmetic in terms of the wordlength and fractional wordlength for those familiar with Simulink and Matlab. Table 2 illustrates the same full precision fixed-point as shown in Table 1 but in terms of VHDL fixed-point declaration. For further information on the usage and capabilities of the VHDL-200x fixed-point package refer to reference [5].

<table>
<thead>
<tr>
<th>Fixed-Point Arithmetic Operation</th>
<th>Wordlength (WL)</th>
<th>Fractional Wordlength (FWL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>Max(A WL, B WL) + 1</td>
<td>Max(A FWL, B FWL)</td>
</tr>
<tr>
<td>A - B</td>
<td>Max(A WL, B WL) + 1</td>
<td>Max(A FWL, B FWL)</td>
</tr>
<tr>
<td>A * B</td>
<td>A WL * B WL</td>
<td>A FWL</td>
</tr>
<tr>
<td>A rem B</td>
<td>Min(A WL, B WL)</td>
<td>A WL + B WL</td>
</tr>
<tr>
<td>Signed A / B</td>
<td>Max((A WL - A FWL - 1), (B WL - B FWL - 1)) + 1</td>
<td>Max(A FWL, B FWL)</td>
</tr>
<tr>
<td>Signed A mod B</td>
<td>A WL % B WL</td>
<td>A FWL</td>
</tr>
<tr>
<td>Signed Reciprocal(A)</td>
<td>-A WL + 1</td>
<td>-A FWL</td>
</tr>
<tr>
<td>Signed Abs(A)</td>
<td>A WL + 1</td>
<td>A FWL</td>
</tr>
<tr>
<td>-A</td>
<td>A WL + 1</td>
<td>A FWL</td>
</tr>
<tr>
<td>Unsigned A / B</td>
<td>A WL + B WL</td>
<td>A WL + B WL</td>
</tr>
<tr>
<td>Unsigned A mod B</td>
<td>A WL % B WL</td>
<td>A WL + B WL</td>
</tr>
<tr>
<td>Unsigned Reciprocal(A)</td>
<td>-A WL + 1</td>
<td>-A FWL</td>
</tr>
</tbody>
</table>

Table 1 - Full Precision Fixed-Point Arithmetic

### Table 2 - Full Precision VHDL-200x Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>Max(A left, B left) + 1 downto Min(A right, B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>A - B</td>
<td>Max(A left, B left) + 1 downto Min(A right, B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>A * B</td>
<td>(A left+B left+1) downto (A right+B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>A rem B</td>
<td>Min(A left, B left) downto Min(A right, B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>Signed A / B</td>
<td>(A left-B right-1) downto (A right-B left)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>Signed A mod B</td>
<td>Max(A left, B left) downto Min(A right, B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>Signed Reciprocal(A)</td>
<td>-A right downto (-A left-1)</td>
<td>A right</td>
<td>A left</td>
</tr>
<tr>
<td>Signed Abs(A)</td>
<td>(A left+1) downto A right</td>
<td>A left+1</td>
<td>A right</td>
</tr>
<tr>
<td>-A</td>
<td>(A left+1) downto A right</td>
<td>A left+1</td>
<td>A right</td>
</tr>
<tr>
<td>Unsigned A / B</td>
<td>(A left-Bright) downto (A right-B left-1)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>Unsigned A mod B</td>
<td>A left downto Min(A right, B right)</td>
<td>A left</td>
<td>A right</td>
</tr>
<tr>
<td>Unsigned Reciprocal(A)</td>
<td>(-A right+1) downto -A left</td>
<td>A right</td>
<td>A left</td>
</tr>
</tbody>
</table>

### REFERENCES


[12] Land, George; Jarman, Beth; Breakpoint and Beyond: Understanding and Shaping the Forces of Change; 1990


A Fixed Point VHDL Library for a High Efficiency Reconfigurable Radio Design Methodology

Scott Hoy – Honeywell TSI
scott.d.hoy@gsfc.nasa.gov

Marco Figueiredo – MEI Technologies
marco@meieses.com

Military and Aerospace Applications of Programmable Logic Devices (MAPLD) Conference
September 2006
Summary

- FPGAs bring major changes to the application design paradigm. Higher performance comes with greater design complexity.
- Managing complexity through an efficient design methodology.
- VHDL as the “netlist” for higher level tools.
- Component library increases design efficiency.
- Reconfigurable radio component library description
- An example design.
FPGAs and the Paradigm Shift in Computer Design

- Phase 1 – Formation
  - Divergence of ideas
  - Invention of new paradigm

- Phase 2 – Consolidation
  - Perfection of established paradigm
  - Rejection of non-fitting parts

- Phase 3 – Transformation
  - A bifurcation appears and a new phase 1 is created
  - A process of reinvention starts in a more complex level.
  - The new paradigm is covered by the current successful one.
  - It is clouded by its own state of disorder.
  - It emerges suddenly.
Managing Complexity Through an Efficient Design Methodology

- Communications application development process
- Document based flow
  - Requirements Document
  - Detailed Design Description
  - Test Plan
  - System Characterization
- Cross domain interfaces definition
Component Libraries
Increase Design Efficiency
**VHDL Fixed-Point Library Objectives**

- Improve the conversion time of translating a Simulink fixed-point model to a VHDL fixed-point model that is both bit and cycle accurate.
- Emphasis on capturing the look-and-feel of Simulink inside the current VHDL language definition – mathematical notation is king.
- Provide complete and absolute control over the VHDL code design of a given algorithm – nothing is hidden from the designer.
- Promote design reuse by creating highly generic synthesizable VHDL components within the current VHDL standard.
- Minimize re-inventing the wheel.
Design Domains Limitations

- VHDL Language Standard
  - Limited generic abstraction (Example: no fully generic arrays or records).
  - Limited data types to simplify higher level abstraction.
  - Limited by the synthesis tool – how much of the VHDL standard does it support?

- Matlab/Simulink
  - Not a traditional logic design tool.
  - Block diagram design abstraction.
  - Supports fixed-point analysis.
  - Model based design approach – similar to VHDL.
Design Issues with Simulink to VHDL Code Generation Tools

- Do not support native Mathworks data types, toolboxes, or blocksets.
  - Example: multi-dimensional arrays, fixed-point data type, Stateflow, simulation accelerator, and validation and verification tools.

- Code generation tools are done by 3rd party, not by the Mathworks. This leads to "blockset hell".

- Designer has limited to no control over how the code is generated.

- Generated code is usually not intended to be viewed outside the code generation process but treated as a "blackbox", which leads to verification limitations.
**VHDL Improvements in VHDL-200x**

- Improved generics and data types to support higher levels of abstraction (similar to C++ templates).
- Addition of a synthesizable fixed-point data type package with supporting synthesizable fixed-point math operations.
- Addition of a synthesizable floating-point data type package with supporting synthesizable floating-point math operations.
Fixed-Point Notation

Signed Fixed-Point (Two's Complement):

<table>
<thead>
<tr>
<th>(5)</th>
<th>(4)</th>
<th>(3)</th>
<th>(2)</th>
<th>(1)</th>
<th>(0)</th>
<th>(-1)</th>
<th>(-2)</th>
<th>(-3)</th>
<th>(-4)</th>
<th>(-5)</th>
<th>(-6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

2^5 2^4 2^3 2^2 2^1 2^0 2^-1 2^-2 2^-3 2^-4 2^-5 2^-6

Binary Point

Unsigned Fixed-Point:

<table>
<thead>
<tr>
<th>(5)</th>
<th>(4)</th>
<th>(3)</th>
<th>(2)</th>
<th>(1)</th>
<th>(0)</th>
<th>(-1)</th>
<th>(-2)</th>
<th>(-3)</th>
<th>(-4)</th>
<th>(-5)</th>
<th>(-6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
</tbody>
</table>

2^5 2^4 2^3 2^2 2^1 2^0 2^-1 2^-2 2^-3 2^-4 2^-5 2^-6

Binary Point

Signed and Unsigned Fixed-Point Format
s = sign bit, i = integer bit, f = fractional bit
wordlength, wl = 12-bits
integer wordlength, iwlsigned = 5-bits, iwluunsigned = 6-bits
fractional wordlength, fwl = 6-bits
VHDL-200x
Fixed-Point Data Type

- VHDL Signed Fixed-Point Declaration
  - signal x : sfixed(5 downto -6);
- VHDL Unsigned Fixed-Point Declaration
  - signal x : ufixed(5 downto -6);
- VHDL Fixed-Point Attributes
  - Wordlength = x'length = 12
  - Signed Integer Wordlength = x'high or x'left = 4
  - Unsigned Integer Wordlength = x'high+1 or x'left+1 = 5
  - Fractional Wordlength = abs(x'low) or abs(x'right) = 6
  - Signed Real Value: -10.40625
  - Unsigned Real Value: 53.59375
VHDL-200x Fixed-Point Mapping to Simulink Fixed-Point

- Simulink Fixed-Point Notation:
  - Signed: sfix(wl), output scaling: $2^{-fwl}$
  - Unsigned: ufix(wl), output scaling: $2^{-fwl}$

- Matlab Fixed-Point Toolbox:
  - $q_x =$
  - $\text{fi}(x, \text{sign}, \text{wl}, \text{fwl}, 'RoundMode', 'floor', 'OverflowMode', 'saturate'),$ where sign = true|false for signed|unsigned.

- VHDL Fixed-Point Notation
  - Signed: sfixed((wl−fwl−1) downto −fwl)
  - Unsigned: ufixed((wl−fwl−1) downto −fwl)
**VHDL-200x Fixed-Point Mapping to Simulink Fixed-Point (continued)**

- **Fixed-Point Overflow Mode**
  - VHDL and Matlab/Simulink support saturation and wrap.
  - VHDL: Use constants `fixed_saturate` and `fixed_wrap` that are defined in the fixed-point package to denote fixed-point saturation or wrapping.

- **Fixed-Point Round Mode**
  - VHDL only supports truncation or rounding.
  - VHDL: Use constants `fixed_truncate` and `fixed_round` that are defined in fixed-point package to denote fixed-point saturation or wrap.
  - Matlab/Simulink: Use rounding modes floor for truncation and nearest for rounding. Remaining rounding options are not supported in VHDL fixed-point.
# Full Precision

## Fixed-Point Arithmetic

<table>
<thead>
<tr>
<th>Fixed-Point Arithmetic Operation</th>
<th>Wordlength (WL)</th>
<th>Fractional Wordlength (FWL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B</td>
<td>Max(A&lt;sub&gt;WL&lt;/sub&gt;, B&lt;sub&gt;WL&lt;/sub&gt;) + 1</td>
<td>Max(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
</tr>
<tr>
<td>A - B</td>
<td>Max(A&lt;sub&gt;WL&lt;/sub&gt;, B&lt;sub&gt;WL&lt;/sub&gt;) + 1</td>
<td>Max(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
</tr>
<tr>
<td>A * B</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + B&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>A&lt;sub&gt;FWL&lt;/sub&gt; + B&lt;sub&gt;FWL&lt;/sub&gt;</td>
</tr>
<tr>
<td>A rem B</td>
<td>Min(A&lt;sub&gt;WL&lt;/sub&gt;, B&lt;sub&gt;WL&lt;/sub&gt;)</td>
<td>Min(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
</tr>
<tr>
<td>Signed A / B</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + B&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>A&lt;sub&gt;FWL&lt;/sub&gt; + B&lt;sub&gt;WL&lt;/sub&gt; * 2&lt;sup&gt;-FWL&lt;/sup&gt;</td>
</tr>
<tr>
<td>Signed A mod B</td>
<td>Max((A&lt;sub&gt;WL&lt;/sub&gt;-2&lt;sup&gt;-FWL&lt;/sup&gt;), (B&lt;sub&gt;WL&lt;/sub&gt;-2&lt;sup&gt;-FWL&lt;/sup&gt;)) + 1</td>
<td>Max(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
</tr>
<tr>
<td>Signed Reciprocal(A)</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + 1</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; - A&lt;sub&gt;FWL&lt;/sub&gt;</td>
</tr>
<tr>
<td>Signed Abs(A)</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + 1</td>
<td>A&lt;sub&gt;FWL&lt;/sub&gt;</td>
</tr>
<tr>
<td>-A</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + 1</td>
<td>A&lt;sub&gt;FWL&lt;/sub&gt;</td>
</tr>
<tr>
<td>Unsigned A / B</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + B&lt;sub&gt;WL&lt;/sub&gt;</td>
<td>A&lt;sub&gt;FWL&lt;/sub&gt; + B&lt;sub&gt;WL&lt;/sub&gt; - B&lt;sub&gt;FWL&lt;/sub&gt;</td>
</tr>
<tr>
<td>Unsigned A mod B</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt;-A&lt;sub&gt;FWL&lt;/sub&gt; + Max(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
<td>Max(A&lt;sub&gt;FWL&lt;/sub&gt;, B&lt;sub&gt;FWL&lt;/sub&gt;)</td>
</tr>
<tr>
<td>Unsigned Reciprocal(A)</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; + 1</td>
<td>A&lt;sub&gt;WL&lt;/sub&gt; - A&lt;sub&gt;FWL&lt;/sub&gt; * 2&lt;sup&gt;-FWL&lt;/sup&gt;</td>
</tr>
<tr>
<td>Fixed-Point Arithmetic Operation</td>
<td>VHDL Full Scale Range</td>
<td></td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>A + B</td>
<td>Max(A'left,B'left)+1 downto Min(A'right,B'right)</td>
<td></td>
</tr>
<tr>
<td>A - B</td>
<td>Max(A'left,B'left)+1 downto Min(A'right,B'right)</td>
<td></td>
</tr>
<tr>
<td>A * B</td>
<td>(A'left+B'left+1) downto (A'right+B'right)</td>
<td></td>
</tr>
<tr>
<td>A rem B</td>
<td>Min(A'left,B'left) downto Min(A'right,B'right)</td>
<td></td>
</tr>
<tr>
<td>Signed A / B</td>
<td>(A'left-B'right+1) downto (A'right-B'left)</td>
<td></td>
</tr>
<tr>
<td>Signed A mod B</td>
<td>Max(A'left,B'left) downto Min(A'right,B'right)</td>
<td></td>
</tr>
<tr>
<td>Signed Reciprocal (A)</td>
<td>-A'right downto (-A'left-1)</td>
<td></td>
</tr>
<tr>
<td>Signed Abs (A)</td>
<td>(A'left+1) downto A'right</td>
<td></td>
</tr>
<tr>
<td>-A</td>
<td>(A'left+1) downto A'right</td>
<td></td>
</tr>
<tr>
<td>Unsigned A / B</td>
<td>(A'left-B'right) downto (A'right-B'left-1)</td>
<td></td>
</tr>
<tr>
<td>Unsigned A mod B</td>
<td>A'left downto Min(A'right,B'right)</td>
<td></td>
</tr>
<tr>
<td>Unsigned Reciprocal (A)</td>
<td>(-A'right+1) downto -A'left</td>
<td></td>
</tr>
</tbody>
</table>
Design Tools Utilized

- The Mathworks Matlab and Simulink (Version 2006a)
  - Matlab Toolboxes: Signal Processing Toolbox, Communications Toolbox, Control System Toolbox, Filter Design Toolbox, Fixed-Point Toolbox, Wavelet Toolbox
  - Simulink Blocksets: Signal Processing Blockset, Communications Blockset, Simulink Accelerator, Simulink Control Design, Simulink Fixed-Point, Simulink Verification and Validation, Stateflow, Link for ModelSim
- Real-Time Operating System
  - Wind River VxWorks
- ANSI/ISO C/C++ Compiler
  - GNU C/C++ Compiler
Design Tools Utilized (cont.)

- HDL Simulation
  - Mentor Graphics ModelSim PE/LE/SE (Version 6.2b)
- HDL Design Management and Version Control
  - Mentor Graphics HDL Designer (Version 2005.3)
- HDL Synthesis
  - Synplicity Synplify Pro (Version 8.6.1)
- FPGA Implementation Tools
  - Xilinx ISE (Version 8.2)
VHDL Fixed-Point Library Organization

- Library is partitioned into three separate VHDL libraries.
  - Fixptlib: fixed-point component library
  - Slvlib: standard logic vector (slv) component library
  - Applib: application component library
- Library is written as generic VHDL that is designed to guide the synthesis tool in inferring the correct logic to maximize target technology resources.
- Pipelining has been reduced to a generic parameter entry on all components.
- Allows full use of the Matlab/Simulink toolset.
- Library components attempt to mimic Simulink block library as much as possible within the definition of the current VHDL standard.
- Utilizes the fixed-point data type package that will be added to VHDL with the official release of IEEE VHDL-200x.
Fixed-Point Component Library

- Real/Complex Math Components
  - Adders, Subtractors, Multipliers, Divider, etc.

- Multi-Rate Components
  - Variable downsample and upsample components.
  - Variable decimating and interpolating CIC filters.

- Memory Real/Complex Components
  - Unit delays, variable unit delays, RAM, ROM, sine and cosine ROM’s.
Fixed-Point
Component Library (continued)

- Filter Real/Complex Components
  - FIR Filters – symmetric, non-symmetric, and anti-symmetric including even and odd versions.
  - IIR Filters – second order sections direct form 1 and 2

- Communication Components
  - NCO’s, I/Q vector modulators, 2\textsuperscript{nd} order DPLL loop filters, bi-phase coding, symbol synchronizers, Costas loop, pseudo-random number generators.

- Control Logic Components
  - Comparators and counters.
Standard Logic Vector (SLV) Component Library

- Memory components.
  - RAM, ROM, etc.

- Basic logic components.
  - flip-flops, registers, comparators, etc.

- Communication level converters components.
  - NRZ-L, NRZ-M, NRZ-S.

- Clock divider components.
Application Component Library

- Multi-Rate BPSK Demodulator
- Multi-Rate QPSK Demodulator
- NASA Ground Network Demodulator
- TDRSS Direct Sequence Spread Spectrum Despreader with Multi-Rate Demodulator, including all the TDRSS spreading codes.
VHDL Fixed-Point Component Library Design Example

- Part 1. Demonstrate a fixed-point library primitive, i.e. a root level component block.
- Part 2. Demonstrate the fixed-point library component that is constructed out of the library primitives.
- Part 3. Demonstrate a fixed-point application that is constructed out of the library components.

*Switch the View to Notes Page for a description of the slides.*
**Part 1. Fixed-Point Library Primitive**

**VHDL Fixed-Point Adder Entity:**

```vhdl
entity sfix_add is
    generic(
        wl : natural := 8; fwl : natural := 6;
        RoundMode : boolean := fixed_truncate;
        OverflowMode : boolean := fixed_wrap;
        pd_i_reg : natural := 0;
        pd_o_reg : natural := 1);
    port( x : in sfixed;
          y : in sfixed;
          z : out sfixed((wl-fwl-1) downto -fwl);
          clk : in std_logic);
end entity sfix_add;
```

**Simulink Adder Mask:**

![Simulink Adder Mask](image-url)
Generate Combinatorial Adder

-- Instantiate Combinatorial Adder
comb_add_generate :
  if (pd_reg = 0) and (pd_oreg = 0)
  generate
    comb_mult : process(x,y)
      begin
        z <= resize((x + y),z'_high,z'_low,OverflowMode,RoundMode);
      end process comb_mult;
  end generate comb_add_generate;
Generate Adder with Registered Input

-- Instantiate Input Adder Registers
reg_input_generate :
  if (pd_iereg > 0) and (pd_oreg = 0)
genenerate
    z <= resize((xReg(xReg'length-1) +
                 yReg(yReg'length-1)), z'high, z'low, OverflowMode, RoundMode);
  reg_input : process(clk)
    begin
      if rising_edge(clk) then
        xReg(0) <= x;
        yReg(0) <= y;

        delay_in : for i in 1 to pd_iereg-1 loop
          xReg(i) <= xReg(i-1);
          yReg(i) <= yReg(i-1);
        end loop delay_in;
      end if;
    end process reg_input;
  end generate reg_input_generate;
Generate Adder with Registered Output

-- Instantiate Output Adder Registers
reg_output_generate :
  if (pd_lreg = 0) and (pd_oreg > 0)
  generate
    add <= resize((x + y),add'high,add'low,OverflowMode,RoundMode);
    z <= zReg(zReg'length-1);
    reg_output : process(clk)
      begin
        if rising_edge(clk) then
          zReg(0) <= add;
          delay_out : for i in 1 to pd_oreg-1 loop
            zReg(i) <= zReg(i-1);
          end loop delay_out;
        end if;
      end process reg_output;
  end generate reg_output_generate;
-- Instantiate Input and Output Adder Registers
reg_io_generate :
  if (pd_i_reg > 0) and (pd_o_reg > 0)
    generate
      add <= resize((xReg(xReg'length-1) +
                      yReg(yReg'length-1)),add'high,add'low,OverflowMode,RoundMode);
      z <= zReg(zReg'length-1);
    reg_io : process(clk)
      begin
        if rising_edge(clk) then
          xReg(0) <= x;
          yReg(0) <= y;
          zReg(0) <= add;

          delay_in : for i in 1 to pd_i_reg-1 loop
            xReg(i) <= xReg(i-1);
            yReg(i) <= yReg(i-1);
          end loop delay_in;

          delay_out : for i in 1 to pd_o_reg-1 loop
            zReg(i) <= zReg(i-1);
          end loop delay_out;
        end if;
      end process reg_io;
    end generate reg_io_generate;
Part 2. Fixed-Point Library Component

VHDL Fixed-Point Non-Symmetric FIR Filter Entity:

```vhdl
entity sfix_fir_filter_nonsym is
    generic(
        wr : natural := 16; fw : natural := 14;
        pr : natural := 20; pf : natural := 16;
        sm : natural := 20; sf : natural := 16;
        RoundMode : boolean := fixed_truncate;
        OverflowMode : boolean := fixed_wrap;
        pd_mul_ireg : natural := 1;
        pd_mul_orreg : natural := 1;
        pd_add_orreg : natural := 1);
    port( din : in sfixed;
          dout : out sfixed((wr-fw)-1 downto -fw);
          coef : in vector_sfix16f14;
          ce_in : in std_logic;
          ce_out : out std_logic;
          clk : in std_logic;
          srst : in std_logic);
end entity sfix_fir_filter_nonsym;
```

Simulink Non-Symmetric FIR Filter Mask:
Part 3. Fixed-Point Library Application Design Process

- Math equation(s) to define a high level system floating point model.
- Refactor high level system model to architecture floating point system model.
- Refactor and quantize architecture floating point system model to a fixed-point system model.
- Translate the fixed-point system model to a VHDL fixed-point system model.
- Verify VHDL fixed-point system model to original fixed-point system model via co-simulation.
- Integrate in reconfigurable radio platform.
- Software and hardware designers should be involved in the architecture design to help aid in design partitioning, configuration and control.
Application: Interpolation Pulse Shaping Modulation Filter

- Algorithm Requirements:
  - $f_s = 4R_s$, where $f_s$ = sampling rate and $R_s$ = symbol rate
  - Bit rate is related to symbol rate by $R_b = \log_2(M)R_s$.
  - Support QPSK ($M = 4$), 8-PSK ($M=8$), 16-PSK ($M=16$), 16-QAM ($M=16$), 16-APSK ($M=16$).
  - Pulse shaping filter is a square-root raised cosine with a roll-off factors of $\alpha = 0.35$ and $\alpha = 0.5$.
  - Filter order of 31-taps (32 coefficients)
  - Target maximum symbol rate of 100 Mega-Symbols per second.
Development of System Equations

- Main System Equation:
  - $y_c[n] = x_c[n] * h_c[n]$
  - Note subscript $c$ denotes complex variables, $(I + jQ)$ where $j = \sqrt{-1}$, and $*$ denotes convolution.

- FIR Filter Equation:
  $$H(z) = \sum_{n=0}^{N-1} h[n]z^{-1}$$

- Interpolation Polyphase FIR Filter Equation:
  $$H(z) = \sum_{r=0}^{N-1} z^{-r} H_r(z^N)$$
Development of System Equations (continued)

- Raised Cosine Frequency Domain Impulse Response:

\[
H_{rc}(f) = \begin{cases} 
T_s, & 0 \leq |f| \leq \frac{(1-\alpha)}{2T_s} \\
\frac{T_s}{2} \left[ 1 + \cos \left( \frac{\pi T_s}{\alpha \left( |f| - \frac{1-\alpha}{2T_s} \right)} \right) \right], & \frac{(1+\alpha)}{2T_s} \leq |f| \leq \frac{(1-\alpha)}{2T_s} \\
0, & |f| \geq \frac{(1-\alpha)}{2T_s}
\end{cases}
\]
Initial Time Domain
FIR Filter Architecture
Time Domain Polyphase FIR Filter Architecture

Bit Stream → Bit To Symbol Mapper and Data Encoder → $H(z)$ → 4:1 MUX DAC

Re$\{y[n]\}$ → LPF → I Ch. Data

Im$\{y[n]\}$ → LPF → Q Ch. Data
Non-Symmetric FIR Filter Architecture
Non-Symmetric LUT FIR Filter Architecture (no multipliers)
Simulink Block Diagram of Time Domain FIR Filter and LUT FIR Filter Architectures
Simulink Block Diagram of Fixed-Point LUT FIR Filter and VHDL LUT FIR Filter Architectures
Floating-Point Time Domain FIR Filter

vs.

Floating-Point LUT FIR Filter Architectures
Floating-Point Time Domain FIR Filter

vs.

Fixed-Point LUT FIR Filter Architectures
VHDL Fixed-Point LUT FIR Filter vs. Fixed-Point LUT FIR Filter Architectures
Future Library Growth

- **Short Term**
  - Further refine current library to make it easier to use.
  - Move library into HDL Designer Pro for library management and version control.
  - Add new components based on current designs and upcoming projects.

- **Long Term**
  - Add primitive components for DSP48 slice and DSP48e slice synthesis inference for Virtex4 and Virtex5 FPGA’s.
  - Add other FPGA support, such as Altera.
  - Add other synthesis tool support.
  - Redo library to be fully VHDL-200x compliant once synthesis tool vendors begin supporting the new standard.
Conclusions

- Pulse shaping modulator filter design running in a lab in terms of a few days versus weeks to months.
- Documentation moves to front seat in order to manage design complexity.
- Component libraries are critical to increase design efficiency.
- Full accountability from model to working hardware leads to complete verification path.
Conclusions (cont.)

- Well integrated multidisciplinary design team required to address increased design complexity.
  - Communications theory/protocols
  - Space communications
  - Space applications of FPGAs
  - Advanced logic design
  - FPGA application design
  - Analog design
  - Digital signal processing
  - VHDL
  - SDR platform programming (RTOS)
  - Software engineering (C, C++, others)