2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

The Application of Formal Technology on Fixed-Point Arithmetic SystemC Designs

Sven Beyer, OneSpin Solutions, Munich, Germany, sven.beyer@onespin-solutions.com
Dominik Straßer, OneSpin Solutions, Munich, Germany, dominik.strasser@onespin-solutions.com
David Kelf, OneSpin Solutions, Boston, USA, dave.kelf@onespin-solutions.com

***Abstract*— The description of abstract algorithmic blocks in SystemC for High Level Synthesis (HLS), has become commonplace. However, block verification is problematic due to the abstraction and coding style of the descriptions, and the lack of adequate tooling. This often forces a partially post-synthesis verification methodology, creating many issues. We will present a native SystemC formal-based approach that targets "loosely-timed" algorithmic requirements, uncommon in HDL verification. An example of automated formal checks and proof evaluations that examine the application of fixed-point arithmetic classes in SystemC will be demonstrated. We will show how this works in a real environment, increasing verification quality.**

I. Introduction

The verification of SystemC abstract, arithmetic blocks tends to incorporate a different set of requirements to RTL verification. At this level, the verification process must ensure that the coded algorithm meets the, often mathematical, specification for all possible usage scenarios, incorporates an appropriate level of accuracy, and can be implemented efficiently. At this early developmental stage, ensuring that algorithm may be effectively and correctly produced using a given level of resources, but without considering the bit level, fully timed nature of the final device, is essential.

The verification required for this phase must consider both the mathematical operation of the algorithm, and its high level implementation against a specification. Although this may be accomplished using a simulation-based approach, the application of formal techniques can greatly improve the ultimate Quality of Results (QoR) and the effort required to achieve those results.

II. Implementation Issues in Fixed-Point Arithmetic

Many algorithms in computer aided design, graphics, and other scientific computations are based on real numbers. A general form of a real number cannot be directly represented in binary digital logic hardware, as an infinite number of bits would potentially be required. This has led to the introduction of Floating-Point numbers as an approximation.

The basic idea of the binary floating-point number representation is that, in addition to a binary fraction called the "mantissa," an integer "exponent" specifies the overall position of the binary point – hence, the binary point "floats". This is an analogy to scientific notation in the decimal format, e.g. $2.51004 \cdot 10^6$. Both mantissa and exponent have a fixed bit width limiting the ultimate magnitude of the number and requiring that the fractional part of the mantissa be rounded off after a specific number of bits. For a given Floating-Point format, numbers close to 0 can be represented accurately with two neighboring numbers being very close to each other, while at the same time, very large numbers with large exponents can be represented accurately with a large distance between two neighboring numbers.

With the release of the IEEE 754-1985 [1] Floating-Point Arithmetic Standard, software developers could rely on their formulas producing the same result on different hardware. The complexity of Floating-Point implementations initially led to the introduction of mathematical co-processers, such as the Intel 8087, in addition to the main CPU. With increased semiconductor device capacity, Floating-Point Units (FPUs) were soon included in the main CPU. However, the infamous Pentium bug [2] illustrates that fully functional FPUs are still very complex and error prone.

For most embedded applications, the cost of actually implementing a full FPU for the required real number computation is prohibitive – both in terms of timing and area. For many purposes, simple binary fractions, so-called Fixed-Point numbers, are good enough to meet the desired computation requirement and do not require expensive hardware.

In Fixed-Point number representation, the binary fraction:
$b_{n-1}b_{n-2} \dots b_1 b_0 . b_{-1}b_{-2} \dots b_{-m}$

with n bits before the binary point and m bits after the binary point represents the value:
$\sum_{-m \le i < n} b_i \cdot 2^i$

b_{n-1}	b_{n-2}	…	b_1	b_0	b_{-1}	b_{-2}	…	b_{-m}
2^{n-1}	2^{n-2}	…	2	1	½	¼	…	2^{-m}

The first digit after the binary point is valued ½, the second ¼, and so on. This format is extended to signed numbers just like signed integer numbers, see Figure 1.

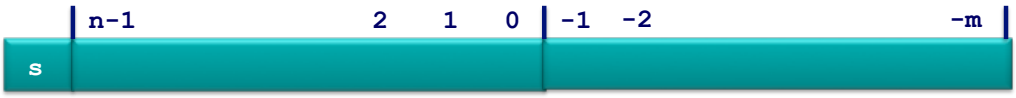

Figure 1 - Representation of a Signed, Fixed Point Binary Number

If the range of numbers used as operands and result is known in advance, Fixed-Point arithmetic is a real alternative to a full-blown Floating-Point numbers, and in some cases, the Fixed-Point arithmetic may even be more accurate. Consider a 32-bit fixed floating number with 1 digit before the decimal point. This format can represent both the numbers 1 and 2^{-31} exactly, just like a single precision Floating-Point number, by making use of its exponent. On the other hand, the sum of these two numbers can only be represented exactly in the Fixed-Point format since it requires 32-bit accuracy to represent. In single precision, Floating-Point the Mantissa does not have enough bits, so the number will be rounded, either to 1 or to $1 + 2^{-23}$.

The major advantage of Fixed-Point numbers over Floating-Point in hardware is in the complexity of the arithmetic unit implementation. Fixed-Point numbers and operators have, therefore, been made available in hardware description languages such as VHDL [3] and SystemC [4] and are generally supported by Synthesis tools. To successfully use this format the range of the numbers needs to be carefully considered throughout the implementation of a mathematical algorithm.

Figure 2 shows an example of this on a generic hardware implementation of an FIR Filter using simple numbers. Given that the coefficients in this filter are numbers between 0 and 1 with a maximum precision of 16 bits, it is possible to vary the bus widths after the application of the coefficients to optimize the filter performance, given the fact that the coefficients are known and will not change. Note that this may lead to also varying the size of the multiplication and summation blocks as well. Of course coefficient selection will be made with the idea of reducing computation effort, but what is the ideal method to test that the ultimate filter will exhibit the correct characteristics while using an optimal level of hardware resources, for all input sequences?

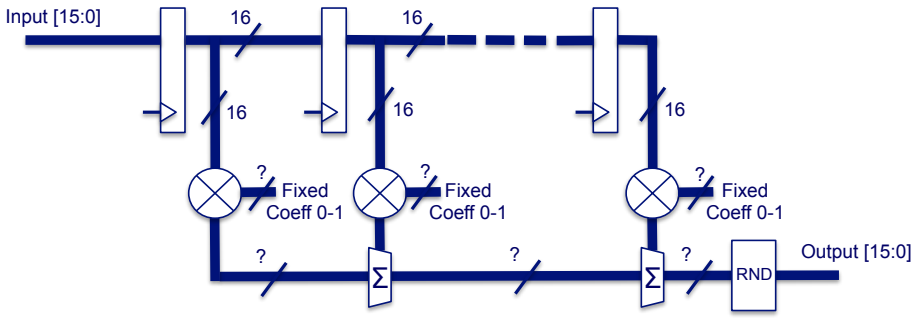

Figure 2 - Generic FIR Filter Implementation

The number of bits before and after the binary point that is required at each arithmetic operation is critical. A lack of bits may lead to incorrect results caused by overflows or an unacceptable loss of precision due to rounding. On the other hand, too many bits may cause a waste of on chip resources leading to excessive transistor usage and power consumption. Any automated help in choosing the proper number of bits before and after the binary point saves engineering time while reducing the risk of errors and optimizing power consumption.

In reality the numbering system for a Signal Processing algorithm will make use of variable width numbers, Complex Numbers with common exponents, and number systems modified to allow for simple multiplication and other arithmetic functions. The calculation of the optimal number widths throughout one of these systems can be very difficult.

III. VERIFICATION OF SYSTEMC DESIGNS

C++ and SystemC are commonly used to implement arithmetic algorithms at an abstract level, such that they may be applied to a High Level Synthesis (HLS) tool. This level may include both code abstraction up to Transaction Level Modeling (TLM), and timing abstraction to a "Loosely Timed" model, i.e. no intermediate timed registers, as defined in the SystemC OSCI standard. The verification of this code usually consists of a software style compilation / execution process, followed by a more thorough test of the resulting, synthesized Verilog or SystemVerilog Register Transfer Level (RTL) representation.

A more rigorous verification of the C++ or SystemC code is desirable. To date, tools have not been available to perform a hardware centric, in-depth test of this code and the algorithm abstraction has been viewed as a software test problem. As these designs become larger and more complex, and are driven by a specification that includes hardware considerations such as bit accuracy, a level of hardware style verification is appropriate at this level to eliminate issues that will be harder to deal with later in the development process.

One way of handling this is to concentrate the verification efforts on the synthesized Verilog code. This is problematic as the code-base is machine generated so typically harder to understand and an order of magnitude larger than the original source. It also contains hardware features that disguise the basic algorithm operation. These issues combine to decelerate simulation performance and complicate problem analysis.

For SystemC blocks targeted at HLS tools, formal tools can be effectively applied to examine particular issues best resolved early in the design flow, before hardware details have been added to the block by the synthesis process. As with RTL code there are a number of automated apps that can be applied to target particular problems at the abstract level. Additionally, assertions that mimic the design specification may be created that more efficiently leveraged at this level of abstraction. It is worth considering how abstract SystemC code fits in a generic verification process.

In standard simulation-based verification, the engineer feeds interesting input sequences into the design under test (in this case the SystemC coded block) and then monitors the outputs for expected values, with the objective of achieving a high degree of code coverage. In general the block is driven using stimulus on the inputs of the DUT. With appropriate verification methodologies, such a simulation-based flow can achieve a high verification quality. However, it is critical that the right input signal sequences are applied to catch issues, thereby creating a significant risk factor.

Formal verification uses an alternative approach. It contains information about all the possible states and transitions a block might execute, and processes questions using this data, thereby eliminating the need to activate specific states with stimulus.

Thus, the engineer picks a scenario for which he would like to see a sequence and leaves it to the formal tool to either find the sequence or prove that such a sequence does not exist. Broadly speaking, such scenarios fall into two main categories: monitors and cover statements.

Monitors are designed to spot unintended behavior, for example an overflow, an array out of bounds access, or a write-write race. As all possible states are tested, formal tools are particularly good at finding corner case input sequences to show a violation of the monitor. Alternatively, a formal tool can also provide the mathematical proof that a monitor can never be violated for any sequence of inputs, as such, creating an exhaustive test that the monitor will never be violated.

Cover statements are about finding input sequences that make certain desired behavior happen. Examples include triggering specific corner cases such as a full FIFO or large and small values in Fixed-Point arithmetic blocks, both very relevant for abstract SystemC blocks.

Cover statements are a very useful tool in early design verification / exploration typical of SystemC algorithm development. For example, a desired behavior may be specified on an output or internal signal set. The formal tool is then used to do the tedious work of extracting an input sequence to produce this scenario, computing the full output behavior. This allows issues to be identified early, simply by inspection of the waveforms on the signals that the tool produces.

With the work of Cimatti, Narasamdya, and Roveri [5] that demonstrates how a number of Software Model Checking (SMC) formal-based approaches may be applied to SystemC designs, and [6], which demonstrates a hardware formal approach, it has been shown that formal verification technology can be applied to SystemC designs. This work has been extended in several directions, including the support of Fixed-Point arithmetic and related automatic checks, including the detection of overflows and redundant bits. Figure 3 shows the OneSpin formal environment with a waveform view depicting fixed float values and value annotation in the SystemC source code of a Finite Impulse Response (FIR) filter.

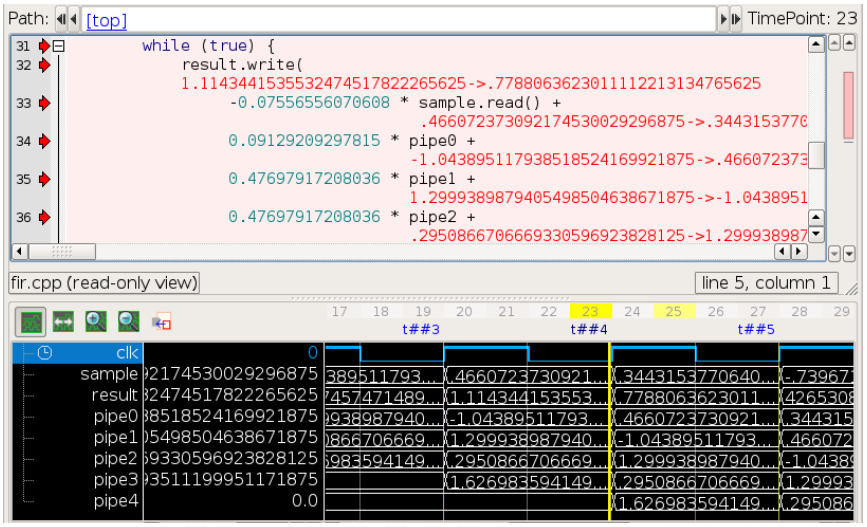

Figure 3: Debugging with Fixed-Point numbers

IV. AUTOMATED FORMAL CHECKS FOR SYSTEMC HLS ABSTRACTIONS

SystemC based designs, similar to other HDL developments, lend themselves to numerous automated formal checks that monitor potential problems. Typical checks could include out-of-bounds array access tests, race condition analysis, dead code checks, and so on. These checks allow the identification of problems as soon as the code has been written instead of later in the flow, where debugging these issues is typically more time consuming.

For Fixed-Point arithmetic processing, a number of unique automatic checks apply. A check for redundant bits at points in an arithmetic datapath might test for whether the Most Significant Bits (MSBs), Least Significant Bits (LSBs), or Carry Bits are actually used.

For signed types, this check may be performed by considering whether the two MSBs can have different values, as shown in Figure 4. If it may be proved formally that the bits always have the same value, the most significant bit is actually proven to be redundant. A similar check may be applied to unsigned Fixed-Point numbers. In this case it is enough to check whether the most significant bit can actually be set to 1. All these tests may be collected together and if redundancy is indicated, used to drive optimizations that are guaranteed to be safe.

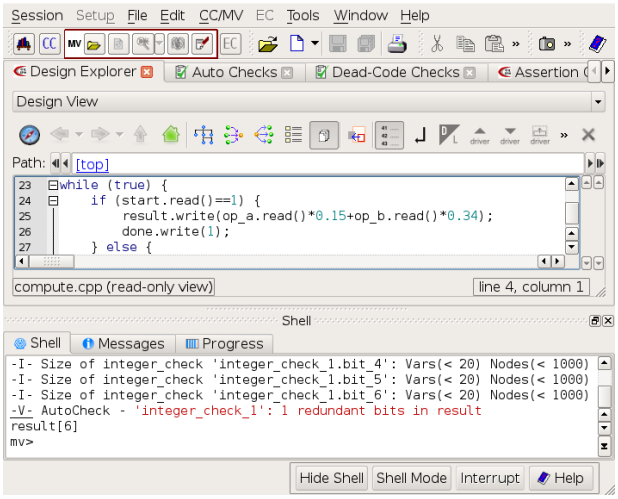

Figure 4 - Example of Redundant Bit Calculation

Overflow checks are also useful at this level, as shown in Figure 5. These checks test whether any possible arithmetic operations can produce a number too large to be represented in the specified data type. Again formal methods may be easily be used to test for overflow given any operational scenario. In a complex arithmetic calculation unit, this can be invaluable.

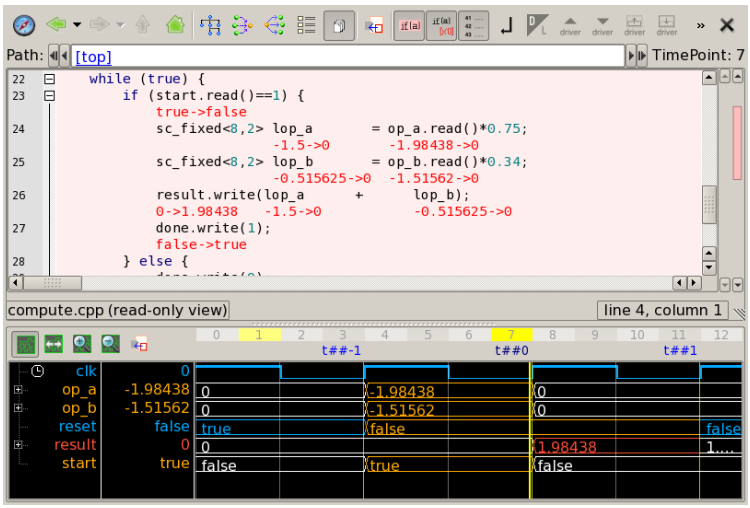

Figure 5 - Example of an Overflow Calculation

In combination, the formal overflow and redundant bit checks together can facilitate the optimal selection of bit widths in an arithmetic calculation block.

While extremely useful, these automated techniques do not address the problem of "sufficient precision," that is, is the accuracy of a represented number sufficient for the modeled algorithm? For example, any Fixed-Point multiplication where the result has the same precision as the operands, will lose half of the relevant bits after the binary point of the exact result, due to rounding. Depending on the algorithm, this may or may not be perfectly acceptable. To understand the effect of rounding through an entire algorithm data flow, user-written assertions can be beneficial.

V. USING ASSERTIONS ON SYSTEMC CODE

Assertions are a well-known concept in C/C++. A simple C assert statement provides a check for assumptions, while a specialized SystemC version "sc_assert" improves upon this with a slightly improved output. These assertions simply evaluate C++ expressions, but do not offer a temporal layer, so cannot handle timed queries or properties. In this sense, they are similar to the VHDL assert statement. They are typically used for checking pre-conditions when entering a C++ method, or post-conditions when returning from a method.

Since these assertions make use of C++ expressions, they can of course make full use of the SystemC Fixed-Point types to check arithmetic properties, such as a temporary variable never taking the value 0:

assert(tmp_result!=0.0)

SystemVerilog Assertions (SVA) do provide a rich set of temporal capabilities, and also have the advantage of being relatively well known. As such we would advocate the use of SVA for SystemC/C++ code. This has the additional advantage of the reuse of these assertions, if written appropriately, both on different SystemC / C++ code blocks as well as in the resulting Verilog code post-synthesis, providing a useful comparison to double check the synthesis process itself.

OneSpin has leveraged SVA for its SystemC capability, as well as C asserts. This is done by using the SystemVerilog "bind" construct to connect a SystemVerilog module containing assertions into a C-design, accessing the signals in the SystemC design by way of hierarchical identifiers, see Figure 6.

```
module fir_check(input [15:0] signed, input clk,
input [15:0] coeffs [x:0], output [15:0] result);

assert property  fir_test (
@(posedge clk)
result = coeffs[0] * signed +
coeffs[1] *  $past(signed,1) +
...
coeffs[x] * $past(signed,x)
// where x = number of FIR stages
);
endproperty
endmodule

bind FIR_Filter fir_check checker_instance (.*);
// where FIR_Filter is the SystemC filter code
```

Figure 6 - FIR Filter Test Assertion

This allows timed assertions to be expressed, for example, on the control path in a SystemC design. Consider a SystemC design that includes a handshaking mechanism that accepts an external request signal, performs some computation, and then sends an acknowledge signal with the computed answer to another block. One risk in such a design is a deadlock situation, e.g. the design stops operating in some way after receiving a request and fails to send the acknowledge. This is a classic case for a simple, yet powerful feature of temporal assertion languages known as "liveness."

In a formal tool, the assertion below will find deadlock scenarios and/or prove the absence of deadlocks. A deadlock is reported to the engineer as a waveform with a loop of input stimuli that can be repeated forever without the design leaving the deadlock, so it actually shows an infinite simulation trace to the engineer:

assert property (systemc.req |-> s_eventually systemc.ack);

For datapath design, SVA has some limitations imposed by a lack in SystemVerilog of the native support for Fixed-Point data types. SystemVerilog is only aware of integers and non-synthesizable doubles. As such in this situation SVA may not be the best choice for abstract SystemC arithmetic blocks, unless the SVA nomenclature is extended.

We have found it necessary to implement a slight extension of the SVA standard. For SystemC signals referenced with hierarchical identifiers, the original type is known from the SystemC source code. As such, the same definition may be simply used in the SVA layer as well, enabling the use of Fixed-Point operators with the respective SystemC semantics in SVA.

Assertions that check for specific coverage are also useful at this level. Leveraging the SystemVerilog Cover Point constructs, various results may be examined. Figure 7 shows a portion an SVA cover point that "covers" a sequence of the values 1.25, 1.75, and 2.625 used in a Fast Fourier Transform (FFT) SystemC design, on the

output signal out_real. The OneSpin ABV tool is then able to automatically find an input sequence of approximately 40 clock cycles that result in the desired sequence of values on the out_real signal.

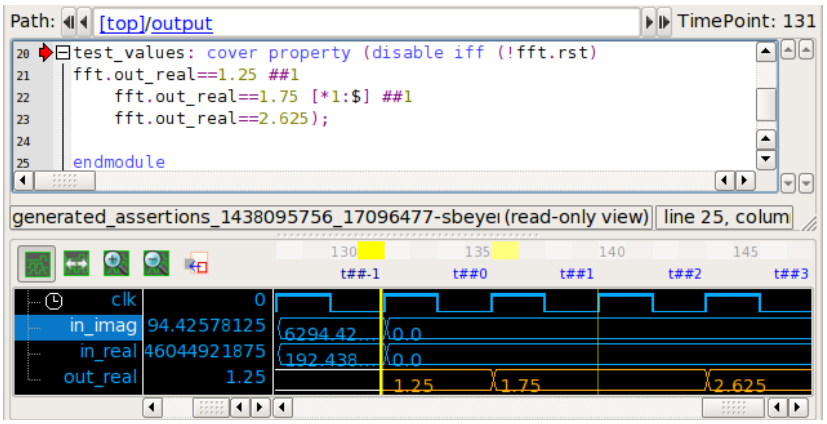

Figure 7: SVA coverage point for sequence of Fixed-Point outputs

VI. RESULTS

Using SystemVerilog assertions, we have been able to demonstrate the types of properties required for abstract, loosely timed, C++/SystemC blocks and how these properties execute in a formal environment. We have also shown how properties may be automatically produced to test for specific issues on various configurations of components coded using SystemC Fixed-Point classes.

Operating on a SystemC DSP example, we show how an arithmetic unit with a range of components and possible arithmetic approaches implemented using the SystemC arbitrary precision, Fixed-Point class, as well as components from other sources (e.g. synthesis supplied IP blocks such as multipliers) may be verified. A broad range of tests can be implemented, that could include, for example, Underflow and Overflow detection on a complex sequence of operations. Although these operations might ultimately require a multi-stage pipeline, at this level of abstraction they simply consist of mathematical blocks with limited timing.

In evaluating this approach on several algorithmic code segments that are employed on a real image processing device, as expected we found that the level of timing abstraction actually allowed the formal tool to run several times faster than an equivalent test on the post synthesized SystemVerilog RTL version of the same code, that included pipeline stages and other lower level detail. This allows for a rapid check of a greater range of high-level properties that mimic the specification of the device, potentially leading to a stronger presence of a formal engine in the overall verification flow.

We were also able to employ several use models on these examples. For example, we were able to easily run complete checks right across the code base to ensure now overflow anywhere in the design, as well as checks focused on specific components that were adapted to ensure that other conditions particular to those components, for example a correct bit reversal operation, could be accomplished. By using constraints on the inputs of the blocks we were able to narrow down the evaluation to common numbers and test the affect of those on the overall system. Overall, the flexibility of the formal approach in this scenario is significant and allows for an effective corner case verification mechanism.

We have also demonstrated how other implementation issues may be analyzed on this high level SystemC code, such as bit redundancy, thus potentially improving the Quality-of-Results (QoR) of the final device. We demonstrated the verification of corner-case type issues, typically difficult to track down using a simulation-based flow leveraging the OSCI SystemC C++ Class Library.

VII. REFERENCES

[1] IEEE 754-2008: Standard for Floating-Point Arithmetic, IEEE Standards Association, 2008
[2] V. Pratt: “Anatomy of the Pentium Bug,” in TAPSOFT’95: Theory and Practice of Software Development (1995)
[3] IEEE 1076 VHDL Language Standard

[4] IEEE 1666 SystemC Standard
[5] A. Cimatti, I. Narasamdya, and M. Roveri: Software Model Checking SystemC, Fondazione Bruno Kessler, Italy
[6] S. Beyer, D. Strasser: Detecting Harmful Race Conditions in SystemC Models Using Formal Techniques, DVCon 2015