Use of C/C++ Models for Architecture Exploration and Verification of DSPs

David Brier
Texas Instruments Inc.
Dallas, TX, USA
+1-214-480-4047
dbrier@ti.com

Raj S. Mitra
Texas Instruments Inc.
Bangalore, India
+91-80-25048132
rsm@ti.com

ABSTRACT

Architectural decisions for DSP modules are often analyzed using high level C models. Such high-level explorations allow early examination of the algorithms and the architectural trade-offs that must be made for a practical implementation. The same models can be reused during the verification of the RTL subsequently developed, provided that various "hooks" which are desirable during the verification process are considered while creating these high level models. In addition, consideration must be given to the qualitative content of these high level models to permit an optimal verification flow allowing for compromise between features of the model and the completeness of the verification. Thus, high quality design and verification are achieved by the use of valid models and the valid use of models. In this paper, we describe our approach and show examples from a typical image processing application.

Category and Subject Descriptors

C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS, Signal processing systems; B.6.3 LOGIC DESIGN, Design Aids, Hardware description languages, Simulation, Verification;

General Terms

Algorithms, Documentation, Design, Languages, Verification.

Keywords

Verification, C/C++, Simulation, Formal, RTL.

1. Introduction

The design process of a DSP system typically starts with development of the algorithm in C language, which includes an exploration of the design space and experimentation with different data structures and algorithms. This algorithm becomes the "valid" golden model on which rests the subsequent steps of design and verification of the DSP module. This model is validated (with reference to the system specification which represents the System Architect's intent) through a set of testcases that measure the algorithm's performance and tests its functionality. Subsequently, the C model is translated into its RTL form, and this RTL is verified against its C reference.

Architecture exploration with the help of high level (C) models is a fairly common practice, because of its ease (of making changes) and speed (of simulation) as compared to doing the same at a lower level of abstraction (RTL). What is not so well understood is how this C model can be used to efficiently verify the lower level RTL and determine its correspondence with the C model. This correspondence is not always straight-forward, due to micro-architectural changes made at the RTL for pipeline efficiency and other purposes. In the absence of a proper verification methodology, the (often manually created) RTL stands the risk of inaccurately representing the high level intent which was signed off as the C golden reference.

This paper discusses the utilization of high level C models throughout the Specification stage, the Development stage and the Verification stage of a design. Specifically, we show that verification hooks built into the model facilitates the verification task, and we illustrate the methodology with examples taken from a typical Image Processing application – the resizing of an image for vertical and horizontal zooming. In general, this methodology is used throughout all designs of this type but there are often specific characteristics which require some small modifications to the flow.

The paper is organized in the following way. Section 2 discusses and compares the roles of the Specification, the C model, and the RTL, and how they are inter-related. Section 3 describes the Resizer design, and this design is used as a case study in subsequent sections. Sections 4 and 5 describe the validation of the C model and the verification of the RTL respectively, and Section 6 discusses the difficulties in verification and the strategies used – both by simulation and formal techniques. Section 7 concludes the paper, and points to future areas.

2. Specification, C model, and RTL

Intent of the design must be documented so that a hard target may be established to which the design must comply. Written specifications are paramount to creating a permanent record of the requirements of the design as well as the evolution of these requirements. These written specifications also serve as a product definition documentation, and is reviewed by the upstream and downstream organizations in the development process.

Writing a good robust C model may be deemed as an executable specification but in the absence of a written specification it will not be sufficient to create a complete and proper design. Specifically, esoteric requirements such as "Image Quality" which lead to a perception of smooth operation by the end user and pleasing visual results in the final image cannot be captured as a C model. The C model therefore has to be validated to conform to the intent of the design as captured in the specification document.

Simply writing a C model without consideration of how it will be used has a significant impact on the development flow. Even though the model may be complete and validated to the specification, it may prove of little use to the development process at worst or have a significant impact on the infrastructure creation unless careful consideration is given to the utilization of the C model in the development process, especially in the verification stages.

It is imperative that the C model be as close to block accurate as possible. This level of model division has several benefits:

- C model matches the functional block diagram.
- C model modules match RTL module partitioning.
- When considering re-use, modifications may be accomplished by substituting the same blocks in the RTL and C models.
- Observation points are more exposed and correlate.

When the high level model is partitioned in the same manner as the block diagram, all of the above listed issues are more simply handled, thus leading to enhanced debugability and higher quality of design re-use. Similarly, a bit-real style of representation enables exact bit comparisons of the output data of the high level model with the output data of the RTL for verification purposes. Bit real representations may also be maintained throughout the model and key points are identified where these bit real representations will be required to enable faster debug of the RTL implementation for arithmetic structures as well as rounding and data concatenation.

Finally, just as a specification document is needed for the overall design, a specification document is needed for the RTL implementation too – for the purpose of creating a high quality maintainable IP. This specification should define the structure that is used in the RTL implementation and will stipulate specific micro-architectures which fulfill the operational requirements. It is important that the correlation between the high level model and the RTL, through some observation points, yield some facilitation of the debug process, and not merely be focused on the ability to generate results which can be used for verification.

3. Overview of the Resizer Module

Although a detailed understanding of the algorithm for performing interpolation and decimation filtering is not absolutely necessary, some overview will aid in understanding the differences between the C model and the RTL model.

Normal filter algorithms have fixed coefficients which are applied to successive input data streams (x) per the following equation. A poly-phase 4 tap filter is implemented by applying up to 4 sets of coefficients across the data set. The data holds for each of the coefficient sets, then progresses to the next shift of data and the cycle is repeated. Thus, x0 shifts to x1, x1 to x2, and x2 to x3, thus creating a shift of the input value.

$$Y = C0*x0 + C1*x1 + C2*x2 + C3*x3 \qquad \text{Eqn (1)}$$

Equation (1) implements a 4 tap filter that, by simply varying the coefficient values, may produce multiple phased output samples. In order to simplify the discussion in the remainder of this paper, the discussion of the filter algorithm will deal with only fixed coefficients.

Figure 1 is a top level block diagram of the C model implementation. It should be observed that each of these blocks has a corresponding block in the RTL block diagram (see Figure 2). The vertical block in Figure 1 corresponds to the four vertical blocks in Figure 2, which are identical to each other and are partitioned for performance reasons. The modules in this design are implemented as bit-real.

The processing steps of the two stages of the Resizer are briefly explained below. The Resizer contains a 4 tap FIR filter for both horizontal filtering and vertical filtering. The horizontal filtering process is examined for the C model and the RTL implementation, to demonstrate the similarities in the data sequence, whereas the vertical filter section is examined to demonstrate a dissimilar data sequence processing. In both cases, the actual implementation of the C model filter differs from that of the RTL and this will also be discussed.

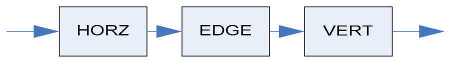

Fig 1: Block Diagram of C Model

3.1 Horizontal Filter Stage

Both the RTL and C models process the data fields in the same sequence, starting with the row and proceeding left to right across the columns. Thus the output data is likewise produced in the same sequence. C code for the filter algorithm might be:

```
for (k = 0; k < num_rows; k++) {     // row index
  for (j = 0; j < num_cols; j++) {     // column index
    for (i = 0; i < 4; i++) {             // filter taps
          y = C[i]*x[i+j][k];
    }
    Yout[j][k] = sat(y);
  }
}
```

The RTL may be implemented as shown below:

```
reg   [15:0]      x0, x1, x2, x3;
always @(posedge clk)
begin
        x3 <= #1 x2;
        x2 <= #1 x1;
        x1 <= #1 x0;
        x0 <= #1 indata;
end
// y = c0*x0 + c1*x1 + c2*x2 + c3*x3;
assign val0 = c0 * x0;
assign val1 = c1 * x1;
assign val2 = c2 * x2;
assign val3 = c3 * x3;
assign y1 = val0 + val1;
assign y2 = val2 + val3;
assign y = y1 + y2;
always @(y)
begin
// put RTL code here to handle saturation …
end
```

Conceptually, the data array for the C model is “rolled” across by the filter code. The original array of data is still present once the filter algorithm has completed processing the entire array of data.

The RTL implementation is designed to minimize gate count and area while meeting performance requirements, thus the data is input to a data pipeline and shifted across the coefficients of the filter. Once the data has exited the x3 register, it is discarded.

Computation of "y" was implemented as a set of assign statements, to allow for the insertion of pipeline stages if required. Coding in this manner allows for insertion of pipeline stages where necessary, with a minimum of effort and without consideration of which tools might be used for synthesis. This coding also makes it simpler to identify the corners of the computation for verification purposes. Data sets may be arranged to cause excursions of the maximum values for the multipliers and adders while maintaining simple input data sets and deterministic coefficients. Ultimately the filter model is verified without regard to it being implemented to perform a low pass filter functionality, i.e. there is no need to pass random data much less to sweep the filter with a range of input frequencies to assure proper RTL implementation.

3.2 Vertical Filter Stage

Vertical filtering in the C model is done starting in column 0, inputting rows 0 to 3, and proceeding through all rows before moving to column 1. Code for this method is:

```
for (k = 0; k < num_cols; k++) {
  for (j= 0; j < num_rows; j++) {
    for (i = 0; i < 4; i++) {
       y = C[i]*x[k][i+j];
    }
     Yout[k][j] = sat(y);
  }
}
```

The above code corresponds to the vertical block in Figure 1. From the code it should be observed that both the input data (x) and the output data (y) are contained in two dimensional arrays. This indicates the necessity of the prior horizontal filter block and the edge enhancement block to have completed their computations prior to the vertical block running.

For the RTL, the implementation of the filter is identical to that of the horizontal filter block. Figure 2 shows four vertical filter blocks which are required to maintain a steady input rate when a 4X interpolation occurs. In this case the four identical filter blocks compute the 4 phase shifted resultants in parallel.

3.3 Feeding the Model

In the Resizer design the format of the input data to the C model was very different than that of the RTL. The C model received data as a two dimensional array of pixel values, whereas the RTL received the data as a 128 bit word. Each symbol was 8 bits wide,

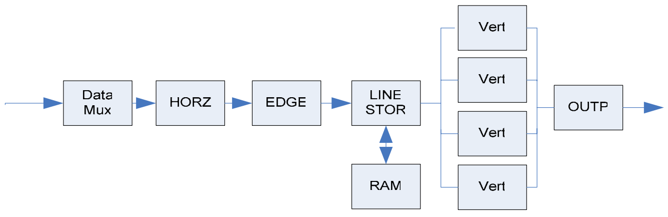

Figure 2: Block Diagram of RTL

therefore the RTL received 16 symbols in each transfer cycle. This data was passed through the data pipe a pixel at a time.

Horizontal operations were performed on a shifted block of 8 bytes in the hardware, whereas the processing in the C model was a simple for-loop iterated through each row of the two dimensional array. Although similarities existed in the migration through each row, some differences did occur in what data values were fed into the computational pipeline, as mentioned below.

Any filter will have edge effects which must be considered. These effects occur on all sides of the array of input data and require a frame of data around the targeted data set. Considering the horizontal filter and only the left most pixel, and the available data set for the C model to be

{d0,d1,d2,d3,d4,d5,…..dn}

If the output computation requires d0 to be in the x0 position of the filter equation, then there must be values provided for x1,x2 and x3. There are many methods for providing this data, and the C model had chosen to mirror the input data, thus providing the following for the computation of the first output value, y0:

{d0,d1,d2,d3} = x0,x1,x2,x3

The manner in which data would be input to the RTL did not permit this method to be used, therefore it was determined that pulling additional data for the 3 pixels on the right and left of each row, and the 3 rows above and below the sample data would be the method of implementation. This caused a modification in the C model to accommodate the additional edge pixels for each row.

4. Validation of the C Model

In the development process, it is the C model that is used primarily for the evaluation of the validity of the algorithms being specified, and it is subsequently leveraged in the verification process as a reference model to which the RTL may be compared. In this validation of the C model, consideration must be given to such issues as round off, quantization noise, and other such computational anomalies which produce artifacts in the final output. These considerations must be accounted for in the same step of the computation as would be done in the RTL model. Thus there may be iterations for the C model as the micro-architecture for the RTL design is derived.

Validation of the C model of the Resizer design was accomplished by applying specific image patterns which are expected to demonstrate anomalies which may be present in the implementation of the zooming algorithms. The outputs are observed and evaluated to be of a level that would be acceptable to the IP end user.

This section is not intended to be a thorough examination of the validation process, but simply to lend an appreciation for the differences between the validation process and verification process.

4.1 Using Standard Patterns for Validation

The test suite applied for model validation is typically determined based on an expert knowledge of the specific domain and its algorithms – in this case, a zooming application. One type of data that can be used for validation of a zoom algorithm are bars and stripes; these are vertical and horizontal in orientation and when viewed easily show anomalies of the zoom algorithm. Equivalent to the step function when evaluating filter algorithms, these patterns may be utilized to view the quality of the interpolation or decimation that is performed on the image.

Depending on the nature of the algorithm and the test suite, the validation is either done quantitatively by comparing with a reference output set, or done subjectively by viewing or comparing with the quality of output generated by an industry standard tool (e.g. Photoshop). In the subjective case, there is no way to automate the validation process.

4.2 Extent of Validation

A "golden model" refers to a model that is completely validated, i.e. there are no more errors to be discovered. The alternative is to perform an adequate bout of validation on the model to achieve a level of assurance that the basic algorithms are correct and thus leaving out corner cases and uninteresting conditions. In the latter case, in subsequent verification stages there is the distinct possibility that a defect may be traced back to the C model itself, and this determination is mediated by the Architectural specification which clearly states the intent and performance requirements of the design.

However, it is not always advisable to elevate the status of the C model to Golden. Aside from the obvious magnitude of effort that might impact the validation of some complex designs, the basic unconstrained nature of some high level models might cause effort to be wasted in the validation of parameters that are impossible to set in the RTL portion of the design. In such cases, a large amount of effort may well be wasted in validation of unreachable design states. Additionally, when using the model as guidance for the implementation of the RTL design, delaying the release of specifications and/or the high level model until completely certain that it is flawless will cause unreasonable delays in the project.

5. Using the C Model for RTL Verification

Two types of C models are typically used, the cycle accurate and the frame accurate models. The cycle accurate models present an accurate representation of the design for each clock cycle. One advantage to the cycle accurate model is that it can flag an error in a simulation on any clock cycles in which that error is detected and by the nature of the model's functionality this is a rather simple process. However, the writer of the cycle accurate model requires more effort to write and more effort and knowledge to tie it into the specific simulator (e.g. through Verilog PLI calls). Close synchronization with the simulator also has a detrimental effect on simulation performance.

On the other hand, imaging and audio applications lend themselves well to frame accurate models, simply because most of the protocols are frame based. Frame accurate models require no knowledge of the tie in of the simulator and are much simpler to write. These models ordinarily input a block of data and output a block of processed data.

5.1 Feeding the Models

The Resizer implements a frame based algorithm which has no memory of previous frames processed. Each frame is unique and stands alone for the zoom processing. One approach to providing input data to a frame based algorithm is to hand the entire frame of data which is to be processed. The data set is delivered to the C model as a two dimensional array which is then operated on by the C model. Configuration parameters are also handed to the C model as an array of values. This approach works well for IP level verification, but alternate methods may be required when the IP is embedded into a sub-system of an SoC.

For the RTL, the input data set can be simply loaded into a RAM and then the outputs accumulated in a RAM and dumped to a file for post processing. Providing data to the Resizer was somewhat more complicated as the data was stored off chip in an SDRAM, and was accessed through 16 byte parallel DMA interfaces. The serialization of this parallel data required another model to translate a RAM access into the DMA transactions required to feed data to the Resizer data pipe.

6. Verification Strategies

Directed and Constrained Random testing have been applied for verification, guided by the coverage goals defined in a detailed verification plan. Proving that the implementation is correct for both models requires characterization of the filters; this can be accomplished using simple patterns (e.g. impulse and step functions) and is best achieved using directed testing methods. In order to produce meaningful coverage for the adders, different data sets and coefficient values were utilized to gain coverage of the full range of the adders.

One consequence of not having a Golden model is that the final arbiter of discrepancies is the specification. Upon observation of a fault in the simulation, the first reaction often is to examine the RTL for a bug in the design. In the Resizer, there were many bugs in the RTL, ranging from data ordering to computational defects. Bugs were discovered in the C model too. The assumption that the C model had been completely tested initially prejudiced assignment of bugs on to the RTL and in some cases caused the C model to be overlooked as a source of a bug, thereby causing the verification process to slow.

6.1 Different Data Processing Schemes

Computing the horizontal filter output values for both models would proceed from the upper left pixel and progress through to the right most pixel of the array, as shown in Figure 3. Although the method of data delivery into the C model and the RTL model were very different, with the C model receiving an entire array of (n x m) pixels as an input to the processing, the manner in which the algorithm progressed through the data was identical to that of the RTL implementation, and the output data results were also identical. That is to say, if the output was the only data stream observed, then there would be no discernable difference between the two models. This simplifies the task of debug during verification, as the output of each model may be tracked directly in a log file for the outputs of those stages.

The Resizer vertical filtering differed considerably, as described earlier (see Figure 4). The RTL model processed data with potentially 4 output values produced in parallel. For interpolation more than one of the parallel filter blocks produces a valid output, as opposed to the rotation of coefficients that would provide interpolation in the horizontal stage, the vertical stage of the RTL produces up to 4 valid phase shifted outputs from each data input set.

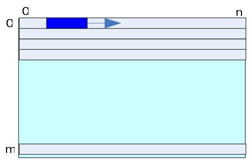

Figure 3: Horizontal Pixel Processing

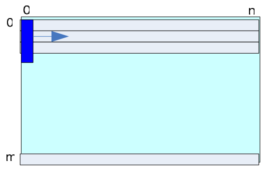

Figure 4: RTL Vertical Pixel Processing

The single most significant difference in the output of the RTL model for the vertical filter stage is in how the output data is sequentially produced. Figure 4 shows the sequence in which data is input to the vertical filter blocks. All four blocks receive the same data inputs. The data inputs start in column 0 with 4 values, and then move on to column 1 for the next 4 values:

(row, col) {0,0;1,0;2,0;3,0}
{0,1;1,1;2,1;3,1}, etc

The first output is rendered as soon as the pixel at location {3,0} is available and the output then proceeds horizontally with each new pixel arriving on row 3. Upon completion of all pixels on row 3, the first pixel of row 4 arrives and it produces the set

{1,0;2,0;3,0;4,0}

and the computations are performed as the lower highest row pixel arrives at the filter block.

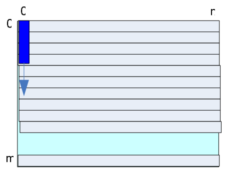

Figure 5: C Vertical Pixel Processing

The C model, on the other hand, produces vertical output in a very different order from the RTL (see Figure 5). Although the phasing is not all that different than that implemented in the RTL the computational ordering of the outputs is. Given that the vertical stage does not run until the horizontal stage has processed all of the input data, the entire array of horizontal output pixels is available for processing. Therefore the first input set is not any different than that of the RTL, but the second and third sets are:

{0,0;1,0;2,0;3,0}
{1,0;2,0;3,0;4,0}
{2,0;3,0;4,0;5,0}

6.2 Debug Requires Commonality of Data

Performing debug using the output of the C model in the Resizer may have been simple for the horizontal stage of the filter because the output of both models was in the same order and all intermediate computations were accomplished in the same sequence in both models. The same may not be said for the vertical stage, the difference between the two models required special consideration in order to enhance debug capabilities. Waveforms were not used except for the simplest computational debug until the specific computation was identified through data comparisons. At that point the waveforms would assist in visualizing problems with the specific RTL structures which actually performed the computation and were normally used to isolate improper structures such as bit rips or data clipping.

Generating debug data from C Model: In this design it was decided to output the results of each stage and the intermediate values as they were produced in the computations. Inserting monitoring in this manner permitted detailed monitoring logs for use in debug while providing on/off control and leaving the original functional code unchanged.

Generating debug data from RTL: It is necessary to produce equivalent output data from both models. It was not practical to observe waveforms to debug the data produced in the RTL simulations, primarily due to operational ordering being different than the C models operations, and hence monitor points were installed at every intermediate computational point to correlate with those in the C model. Output ordering was not dealt with in the RTL model, but the output data from the RTL was re-formatted externally using PERL scripts. This allowed multiple format changes to facilitate debug.

6.3 Formal Verification

Apart from simulations, formal techniques may also be applied to the verification of a design such as Resizer. Formal proofs are by definition exhaustive, and thus provide a guarantee on the quality of the verification. Commercial tools for Formal Verification are mainly in two categories – Model Checking and Sequential Equivalence Checking. In Model Checking, the designer codes the intent of the design in terms of assertions or properties (in languages like PSL or System Verilog Assertions). These properties are then attempted to be proven on the design, i.e. they are the golden reference against which the design is verified. The task of writing the assertions needs a detailed knowledge of the design and is, in every bit, as complex as creating the system level C model and the RTL.

In Sequential Equivalence Checking (SEC) on the other hand, the golden is a model at a higher level of abstraction, but which has considerable differences with the RTL (e.g. latency / pipeline differences, interface differences, etc) which prevent a vanilla equivalence checking to be performed. The C models which are written for system evaluation contain every operation which may be found in the RTL of the design. As discussed in previous sections, the operational ordering may not be identical, but the resultant output is identical to that of the RTL. Thus, the time and effort which had been invested in creation and verification of these C models can be leveraged by applying SEC techniques for verification. The Resizer module seems like an ideal case for SEC, since the C model is already available and verified through simulations; hence without going through the pain of writing the assertions, we can simply verify the RTL against the golden C model.

As with any other method, there are restrictions which apply to the models used as a reference, these are simple to adhere to and the golden C models were readily re-used for the formal verification. The restrictions are very intuitive; they require the C model to have inputs and outputs that can correlate roughly to the inputs and outputs of the RTL, as well as some coding restrictions such as no use of malloc and no pointer aliasing. The two models, the C reference and the RTL, must also be surrounded by wrapper models which provide the tool with the application points for stimulus and observation.

Control of the verification is specified through an input file which provides control of all inputs, specifying the range of the inputs and the types of stimulus that the input supports. Expected latency differences between the two models is also specified in this control file, which helps the formal tool to distinguish between differences which are bugs and differences which are features.

There are always issues in any verification flow; one issue that can be problematic in a formal verification process is the tool's capacity for handling large designs. Solutions for this problem vary from parameterizing the design (e.g. the horizontal and vertical sizes of the Resizer's image) and using lower values for the verification, to using bounded proofs instead of full proofs. The latter strategy can be considered as sufficient if it is known beforehand that the sequential depth (diameter) of the circuit cannot exceed the proof bound achieved.

6.4 Verification Experiences

Release verification for the Resizer was performed using the methods discussed in the bulk of this paper, and the formal techniques were applied post release. Directed testing was primarily used, and it was augmented with directed random testing where a selected set of parameters were permitted to be randomized for testing. This approach was adequate for testing but had some drawbacks which impacted testing, e.g. issues such as compile time and simulation run time as well as capacity issues with very large arrays. A significant issue arose with the discovery of bugs in the testbench, and the nature of the randomness generation which prevented the use of the same seed to generate same stimulus when confirming the fix of a bug. Being incapable of repeating the exact stimulus which exposed the bug leaves a potential hole in the verification process.

Using formal methods on the design showed some of the well known limitations for formal methods, such as state space explosion, but this was manageable in a manner similar to the random test generation (e.g. in both the verification processes, the image size was reduced). On first pass, the formal testing came back with no errors. To ensure that the formal method was capable of showing the faults, faults were inserted into the RTL based on the last bugs located. These bugs were found in a very short time by the formal tool and counter examples were generated which allowed simulation of the bugs in the simulator of choice, thus providing a mechanism to confirm the bug fix.

7. Conclusion

Many current designs utilize High Level C/C++ models in the process of evaluating algorithms which are implemented in SOC designs. As new technologies become available, such as SystemC a migration of those models to these technologies will occur. Considerable effort is spent in the development and evaluation of these models; some of them are often taken to the quality level where they may be considered Golden Models for verification purposes. The value of this effort must be recognized in the RTL development and verification processes, the methods discussed in this paper demonstrate that recognition. Architectural models are leveraged as verification reference models for RTL development and are incorporated into the verification flows providing automated design checking.

Coding methodologies have been developed for the SOC flows which dictate constraints for the coding of RTL, but for successful integration of C/C++ models into a verification flow it is imperative to also have coding guidelines for these models. Partitioning requirements and some structural limitations benefit the verification engineer during debug of test cases. Providing similar points for observations in the C model which correspond to points within the RTL model giving consideration to bit widths as well as operator ordering aid in the correlation of results for verification and debug.

The EDA industry is developing tools which will allow tighter integration of the high level models into simulations, thus allowing co-simulation of the RTL and high level model. When tightly coupled, additional checking may be accomplished allowing automated checking of intermediate values during simulation. Also the application of technologies such as formal methods using the C/C++/SystemC models as reference models to perform formal proofs show great promise moving into the future.

High Level models provide a powerful verification reference when written within a reasonable set of constraints when used in conventional methods as discussed in this paper. As new tools evolve these models may become a larger part of a very powerful verification methodology.

8. References

[1] Tom Schubert, "High Level Formal Verification of Next-Generation Microprocessors", DAC, 2003.

[2] Yves Mathys, Andre Chatelain, "Verification Strategy for Integration 3G Baseband SoC", DAC, 2003.

[3] Yaron Wolfsthal, Rebecca M. Gott, "Formal Verification – Is It Real Enough?", DAC, 2005.

[4] Prem P. Jain, "Cost-Effective Co-Verification Using RTL-Accurate C Models", IEEE International Symposium on Circuits and Systems (ISCAS), 1999, pp 460-463.

[5] Mneimneh M N, Sakallah K A, "Principles of Sequential Equivalence Checking", IEEE Design and Test of Computers, 22(3), May 2005, pp 248-257.

[6] Calypto Design Systems, "RTL Verification without Testbenches", www.calypto.com, 2005.