Stepwise Refinement and Reuse: The Key to ESL

Ashok B. Mehta
TSMC Technology, Inc.
2585 Junction Avenue
San Jose, CA 95134
(408) 678-2722
ashokm@tsmc.com

Mark Glasser
Mentor Graphics, Corp.
46871 Bayside Parkway
Fremont, CA 94538
408-451-5516
mark_glasser@mentor.com

Shabtay Matalon
Mentor Graphics, Corp.
46871 Bayside Parkway
Fremont, CA 94538
408-451-7456
shabtay_matalon@mentor.com

Dan Gardner
Mentor Graphics, Corp.
8005 SW Boeckman Rd
Wilsonville, OR 97070
503-685-7993
dan_gardner@mentor.com

ABSTRACT

For ultra-scale SoC designs that are now commonplace, it has become impractical to use only traditional RTL design and verification techniques. ESL methodologies, used for designing at levels of abstraction above RTL, are instrumental in determining design feasibility, honing requirements, and experimenting with architectures and algorithms to meet functionality as well as performance and power requirements. However, applying ESL techniques requires building abstracted transaction-level models that traditionally do not have a direct path to RTL implementation.

A stepwise refinement and reuse flow is necessary to realize the complete benefits of ESL. It preserves each subsequent modeling investment through the transformation and verification of transaction-level models from their initial highly abstracted representation to fully verified RTL. It utilizes transaction-level models as reference models during RTL verification and reuses the initial TLM platform as a "system level testbench" for downstream implementations.

In this paper we will illustrate the essential elements of a five step refinement flow. The first four steps in the flow have been realized in TSMC's Reference Flow 11 and work is ongoing for reference flow 12. We will briefly show results from reference flow 11 and discuss the work underway for reference flow 12.

Categories and Subject Descriptors

B.5.2 [REGISTER-TRANSFER-LEVEL IMPLEMENTATION]: Design Aids –automatic synthesis, optimization, simulation, verification

General Terms

Design, Verification

Keywords

Electronic System Level, ESL, transaction-level models, TLM

1. INTRODUCTION

Market data suggests that the total number of highly-integrated basic, enhanced, and smart phones will be over 2 billion by 2012. When tablet PCs, iPADs, and other small form factor, connected devices are included, the number is projected to reach over 15 billion by 2015. These devices will require high-end graphic performance in the range of 200 million triangles per second by 2011 and will need to support multiple concurrent software applications. Lastly, all of these devices will require very low power to deliver longer battery life.

These trends combined with transistor density reaching over 40 million transistors per mm2 at 20 nm has caused an explosion of design complexity. However, design productivity is growing by only about 21% per year. This disparity is depicted in Figure 1.

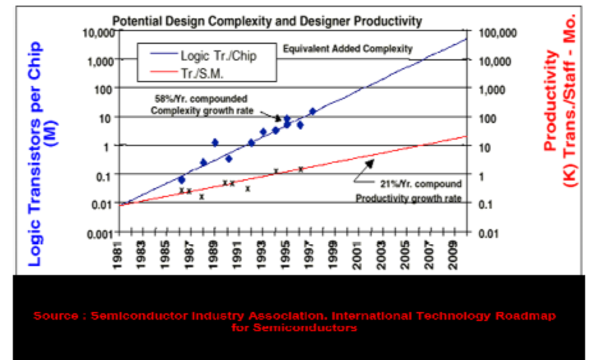

Figure 1. Chip complexity versus design productivity.

Contributing to the increasing gap between complexity versus productivity are current RTL-centric design and verification methodologies that do not scale with the complexity of new devices. Software is impossible to develop on RTL designs and relying on emulated platforms that require synthesizable RTL forces software integration and verification to occur too late in the game.

It has been well established that software development and functional verification are the two most resource and time intensive activities of any hardware design project (Figure 2). RTL-centric methodologies continue to exacerbate this dilemma.

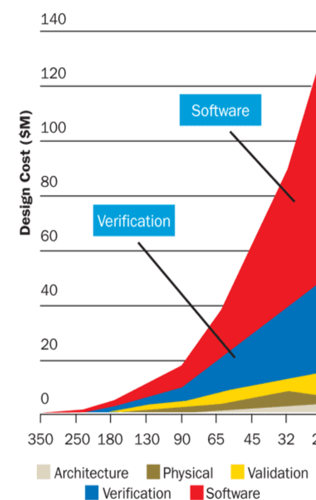

Figure 2. Cost of design tasks per technology.

Methodologies that support abstractions higher than RTL are needed. These electronic system level (ESL) methodologies are the best hope to speed up the initial architectural exploration phase for software/hardware partitioning as well as support early software development, functional verification, and weighing power, performance, and area trade-offs before committing to RTL.

Establishment of the SystemC TLM2.0 standard has paved the way to create modular, reusable, transaction level methodologies at the ESL. But ESL methodologies used in isolation from the RTL flows do not sufficiently advance the move from RTL to ESL, and the benefits therein. A methodology is needed that provides a path from ESL to RTL that reuses and leverages the ESL design and verification work at the RTL.

A stepwise refinement and reuse flow preserves each subsequent modeling investment through the transformation and verification of transaction-level models from their initial highly-abstracted representation to fully verified RTL. It utilizes transaction-level models (TLM) as reference models during RTL verification and reuses the initial TLM platform as a "system level testbench" for downstream implementations.

2. FIVE STEP REFINEMENT FLOW

In this paper we will illustrate the essential elements of a five step refinement flow.

1. Algorithmic
   Models are represented in pure C/C++ algorithmic form and verified using C++ testbench.

2. Transaction Level Model
   The algorithmic models are transformed to transaction-level models and assembled into a transaction-level platform representing the system architecture. The design requirements can be validated and debugged at this stage, and the architecture can be optimized to meet performance and power requirements.

3. Block Level
   The TLM is now implemented in RTL either via High Level Synthesis or coded manually. An OVM testbench is used to verify that the RTL block is functionally correct. The original TLM is used as a reference in the scoreboard to determine functional equivalency. Stimulus is reused from the Algorithmic step.

4. Bus Integration
   The RTL block is connected to a standard bus. Stimulus is reused from the Block Level step. Testbench layering techniques in OVM enable reuse of the stimulus and scoreboards.

5. System Level
   The block is now integrated with other blocks on the bus to form a complete system. Stimulus reused from the Block Level step is included. The system-level testbench and software driven stimulus are reused from the TLM step.

TSMC defined the ESL-Verification step-wise refinement flow requirements as part of its new ESL scope for the Reference Flow 11. Mentor implemented the flow on an IDCT block design using the Vista™, Catapult® C Synthesis, and Questa® tools and methodologies. The example described here was donated by Mentor to TSMC Reference Flow 11.

2.1 Algorithmic Step

The "golden model" and testbench for an Inverse Discrete Cosine Transform (IDCT) model was created in untimed ANSI C++. The IDCT block reconstructs a sequence from its DCT coefficients to bring back the spatial information from a JPEG/MPEG stream. It is a key block in consumer electronic devices for the dissemination and consumption of audio and video, amongst other applications.

The starting point is the IDCT from the International JPEG Group (IJG). While this highly optimized software code is not well suited for high level synthesis, it is an excellent reference model to verify the synthesizable code against.

The C++ model and C++ testbench was considered the "golden source." They were used to verify that the C++ model was correctly exercised by the C++ testbench as compared against the IJG reference. Verifying IDCT at the algorithm level allowed verifying the C++ model and C++ testbench much more quickly than at RTL, more quickly debugging and changing the C++ model and/or testbench than the RTL, and ensuring that the C++ models worked correctly before adding additional timing and communication information further on in the flow.

2.2 Transaction Level Model Step

In this step, TLM 2.0 wrappers are added to the C++ model and the testbench from Step 1. The TLM models are created from the C++ source and assembled to create the TLM platform that is used to validate and debug the TLM models. This allows reuse of the algorithmic C++ model and testbench and faster validation and debug of the IDCT at TLM over RTL. This is also a step towards building virtual platforms. Each TLM component is built and verified on its own before it is used in building the complete SoC virtual platform. A coverage collector is added to evaluate coverage at the TLM level. The algorithmic to TLM model migration, validation and debug is depicted in Figure 3.

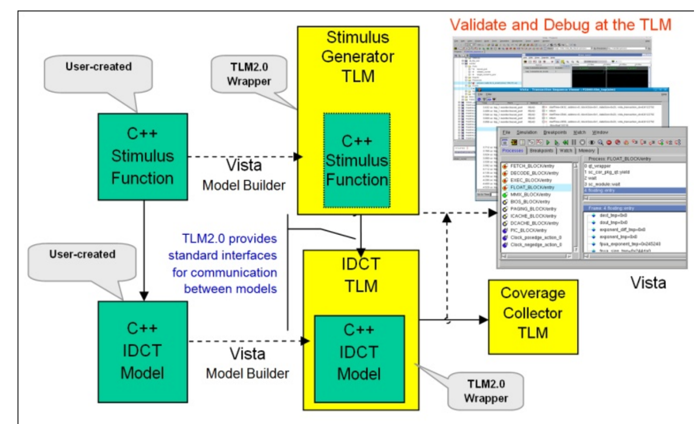

Figure 3. Vista used for Algorithmic-to-TLM model migration, validation, and debug.

The TLM testbench is used to validate the SystemC algorithm. The TLM DUT contains, of course, the algorithm we are validating. The testbench contains two main elements: the stimulus generator that generates a stream of input data and retrieves responses; and the coverage collectors that monitor transaction level activity to determine if the algorithm has been sufficiently exercised.

In this example, the IDCT and Stimulus C++ code is wrapped in a TLM2.0 wrapper using these three steps:

1. For all slave sockets, callback functions were created to enable the model to react to an incoming transaction.

2. For all master sockets, convenience functions were created to initiate outgoing transactions.

3. The behavior of the component was modeled by embedding master convenience functions on the slave callback functions.

The sockets can be either master (to initiate transactions) or slave (to react to transactions), so we defined the master and the slave ports and their width. We then defined the registers, the address for each, and the type of access allowed for each register. Finally we associated a callback reusing the algorithmic model which triggered when the register was read or written. The sockets can instantiate specific protocols. In the absence of a specific protocol, we used the "generic" protocol. Then we proceeded to define the timing relations between transaction boundaries using policies. These policies can be of several types: delay, split, sequential, pipeline buffering, or pipeline interleaving. We have chosen to define sequential and delay policies.

We used a debug and verification environment for obtaining a general view of transactions (READS/WRITE) and their timing relations as well as detailed transaction data structures, status, and phases. To better understand model behavior, both thread execution and event triggering were used. We viewed the transaction in a display waveform viewer and transaction sequences in a transaction sequence view (TSV) window. We also could see the state of each process, its process stack, and local variables, and we could observe when a process was suspended or resumed using the Vista animated process view.

For the IDCT design example we added two coverage collectors, an input coverage model, and an output coverage model. The input coverage model counts the number of times each of the sixty-four words in an input data block have odd parity and even parity. 100 percent coverage was reached when each of the sixty-four words exhibits odd parity and even parity once. There are 64 things that can each have odd or even parity, which means there are a total of 128 bins. The output coverage model counts the number of times each of the sixty-four words in an output data block was saturated at zero (i.e., all zeros), saturated at one (i.e., all ones), or not saturated. For the output coverage model, each of the 64 positions in the output data block can have one of three different saturation modes for a total of 192 bins.

The coverage collectors were implemented as subscribers— devices which connect to analysis ports. The coverage collectors each implemented the write() function in the analysis interface. The *write()* function extracts data, as necessary, from the transaction passed to it and increments the coverage counters accordingly. In SystemVerilog, the counters were implemented using *coverpoints* and *covergroups*. In SystemC, the counters were implemented with local variables.

Once the IDCT TLM was completely validated reaching its functional coverage goal, we had to take an extra step to prepare it for reuse in the OVM RTL verification environment. Currently, the connection between SystemC ports and SystemVerliog ports is done through an interface that supports only TLM1.0 style ports. For reusing the IDTC TLM having TLM2.0 style ports (sockets) as a reference model to the RTL DUT, a translator had to be provided. When the translator receives a TLM1.0 request, it stores the request in memory and creates a generic payload that points to that memory. This payload is written (via *b_transport*) to the TLM2.0 DUT and processed. Then the processed payload is read (again via *b_transport*) and translated into an appropriate TLM1.0 response and sent back.

2.3 Block Level Step

The RTL block is created using high-level synthesis to synthesize the C++ functional core used to define the model of the behavior of the IDCT TLM. Catapult includes built-in support (named, SCVerify) to verify the generated RTL against the C source. It wraps the generated RTL in an automatically created SystemC driver and monitor that allows comparison of the generated RTL code to the synthesizable C++. It generated all the verification models and shell scripts, which allowed us to launch interactive simulation in order to compare and view simulation results.

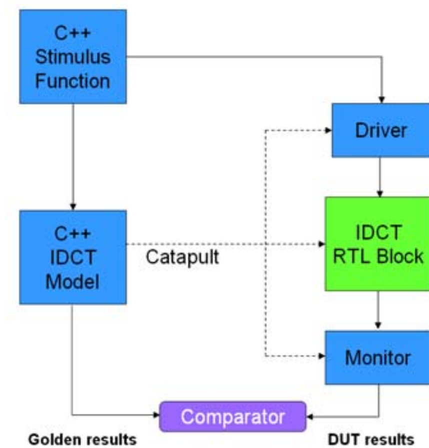

Figure 4. Catapult SCVerify used to produce and verify the IDCT RTL block.

To verify the IDCT DUT we treat the pins on the device as a communication protocol. It is not a standard protocol, but by treating it as such we can build an OVM agent to drive the DUT. The agent is a container that holds a driver, monitor, sequencer, and a coverage collector.

The agent has three interfaces:

1. A sequencer interface
   The sequencer is used to host sequences, behaviors that generate stimulus for the DUT. A reference to the sequencer is used to initiate sequences.

2. An analysis port
   The analysis port makes available the transactions recognized by the monitor to components outside the agent, such as a scoreboard.

3. A virtual interface
   The virtual interface is a reference to an interface object (i.e. the SystemVerilog interface construct) that contains the pins that the testbench uses to talk to the DUT. In this case the virtual interface is the bundle of pins that are on the IDCT DUT.

The other major elements of the block level testbench include:

1. Sequences
   Sequences are behaviors that generate streams of

transactions, also called sequence items, which are used to stimulate the DUT

2. Scoreboard
   The scoreboard is responsible for determining whether or not the DUT provided the correct response for any particular stimulus. The scoreboard contains some machinery for sending requests to the reference model, retrieving responses, and comparing the responses from the reference model with responses from the DUT. The reference model is written in SystemC, so the scoreboard has to communicate across the language boundary.

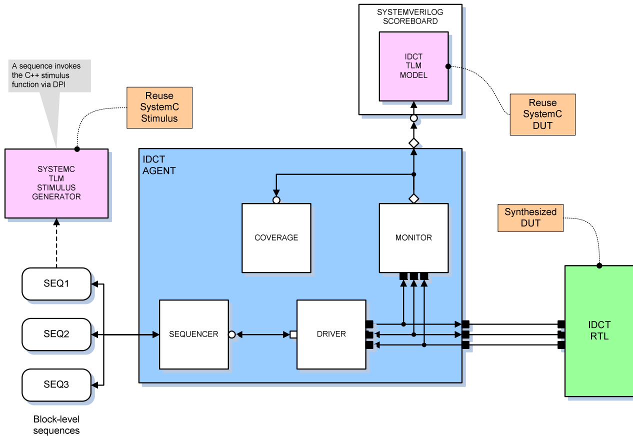

Figure 5. The testbench.

The testbench is a conventional OVM testbench with the exception of the internal construction of the sequences and the scoreboard, as shown in Figure 5. The sequences come from the C++ stimulus reused in the SystemC TLM. In the SystemC model, the C function is linked directly into the model. The stimulus generator provides data to the DUT using SystemC datatypes. Since these datatypes are not available in C, a conversion must be done to convert the data generated by the C stimulus functions into a SystemC datatype for use by the stimulus generator.

In the SystemVerilog/OVM testbench, the same C function is used in the stimulus generation sequence to provide the actual stimulus data. The C function is linked with the SystemVerilog via DPI. Just like with SystemC, a conversion must take place to convert data generated in the C domain to be used in the SystemVerilog domain. This this case, the DPI facility takes care of the conversion.

2.4 Bus Integration Step

As we continue with the stepwise refinement process, the next step is to start integrating the block with the system, as shown in Figure 6. To do that we must be able to connect the IDCT block to an AXI bus, as the system is designed using an AXI bus. We connect an adapter to the IDCT block that enables the block to appear as an AXI slave. The DUT now consists of the AXI slave adapter and the IDCT block. Since the device now has an AXI interface we can drive it with and AXI agent.

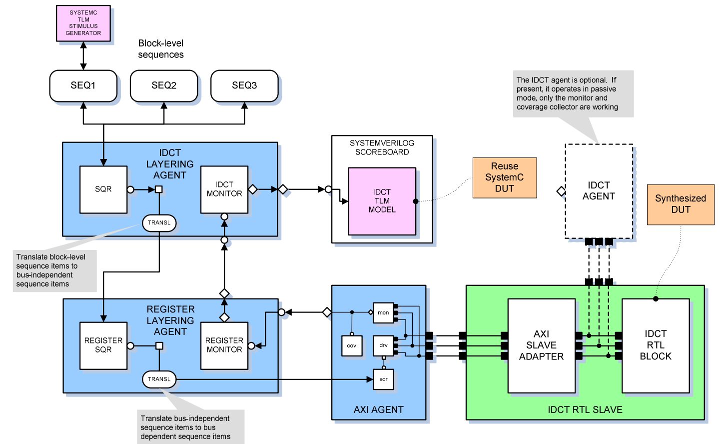

Figure 6. Integrating the block with the system.

The AXI agent we used is from the Mentor MVC library. The MVC library is a collection of verification IP that has SystemVerilog/OVM wrappers so that they can easily be used in OVM testbenches. Like any OVM testbench, each MVC contains a monitor, driver, and sequencer.

In order to use elements from previous steps we have to create layers, agents that talk to other agents, not directly to the DUT. In this case we have two layers, an IDCT layering agent and a register layering agent. The IDCT agent converts IDCT transactions to protocol-independent bus transactions. IDCT transactions are just blocks of numbers that are operated on by the IDCT block. The IDCT agent converts those to read and writes on an abstract bus. In the reverse direction, the IDCT agent converts protocol-independent transactions to IDCT transactions.

Note that the register agent converts the protocol-independent transactions to protocol-dependent transactions, sometimes called concrete transactions. A protocol-independent transaction may say something like “read register A”. The register agent looks up the address of A and converts the transaction to perform a read on the address where register A resides. Finally, the AXI agent will convert the protocol-dependent transaction into pin activity on the RTL bus.

The IDCT agent also has a whitebox connection on DUT. The DUT wrapper contains an interface that connects to the internal IDCT bus. This interface is made available externally. This enables some of the internal activity to be made visible externally. In this example we connect the IDCT agent to the interface. The IDCT agent operates in “passive” mode, meaning that the sequencer and driver are turned off. The monitor and coverage collector remain on. Effectively, the agent operates like a monitor. We instantiate the agent and run it in passive mode instead of just instantiating and connecting a monitor because the agent is the reusable element. It’s much easier to instantiate the agent then to have to understand how to instantiate and connect each internal component.

The sequences generate randomized IDCT input data, pack the data into AXI transactions, and send them off to the AXI agent. The AXI agent converts the AXI transactions to pin-level activity on the AXI bus. The AXI slave, which contains the IDCT block, processes the input data, performs the IDCT computation, and returns the result back to the AXI bus. The AXI agent retrieves the results and makes them available to the sequences. The sequences can get the result by issuing a bus read.

2.5 System Level Step

The final stage in our stepwise refinement process is to connect our IDCT slave to the AXI switch. That enables it to be used in a complete system.

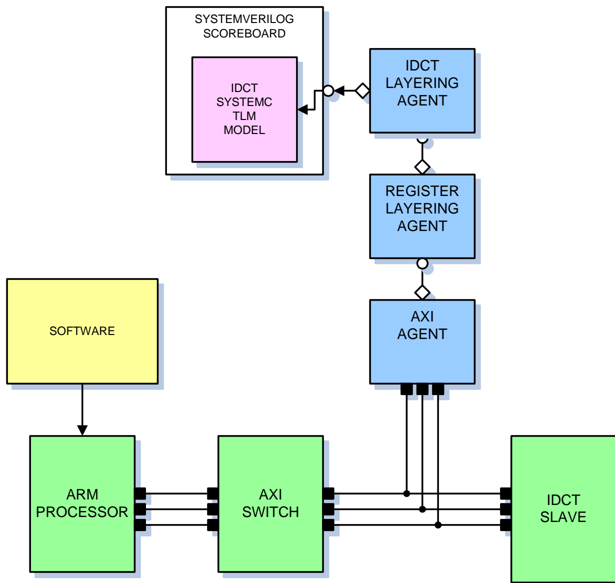

Figure 7. The complete system.

We retain the transactors and the layering from the previous step. We add a processor and whatever other masters and slaves are necessary to form the complete system.

3. CONCLUSION

We have shown that using a stepwise refinement flow we can take advantage of architectural models written in SystemC in the RTL verification flow. At each refinement step we reuse design and verification components from the previous step. This not only saves time by avoiding re-writing models that already exist, but it ensures that the verification results achieved at each step can be recreated at subsequent steps. This provides for greater confidence that the final design is correct and reduces the number of bugs and the entire verification effort.

4. ACKNOWLEDGMENTS

Thanks to Todd Burkholder, Senior Writer, Mentor Graphics for editorial support.