Functional Verification of SiCortex Multiprocessor System-on-a-Chip

Oleg Petlin and Wilson Snyder
SiCortex Inc.
Three Clocktower Place, Suite 210
Maynard, MA 01721,USA
wsnyder@wsnyder.org

ABSTRACT

This paper discusses functional verification of the SiCortex multiprocessor compute node. It is shown that the implementation of reusable verification methodology, applicable at the block- and chip-level, combined with a flexible SystemC testbench design increases the level of verification productivity. Also, it is demonstrated how verification productivity can be improved by using open source verification tools. The simulation approach described in the paper provides a powerful mechanism for controlling the simulation speed, accuracy, and overall verification cost. As a result, the SiCortex verification team was able to find more bugs faster and to start co-verification in early stages of the project development.

Categories and Subject Descriptors

B.6.3 [**Design Aids**]: Verification

General Terms

Design, Verification

Keywords

Functional verification, co-verification, Verilog, SystemC, C++, modeling, coverage, regression testing, code reuse

1. INTRODUCTION

SiCortex cluster computer systems deliver high application performance with less power dissipation and smaller system sizes for low cost. Each system is composed of a large number of six-way Symmetric Multiprocessor (SMP) compute nodes that run the Linux operating system and use the Message Passing Interface (MPI) for communication between nodes. For example, the SC5832 system contains 972 compute nodes connected together in a degree-3 Kautz graph and delivers peak performance of 5.8 teraflops in a compact, low power cabinet [1].

DAC'07, June 4–8, 2007, San Diego, California, USA.

Figure 1 shows a block diagram of the SiCortex compute node. The node is an SMP system-on-a-chip (SOC) with coherent L1 and L2 caches, two DDR-2 memory interfaces, a PCI Express (PCIe) interface to external I/O devices, and a programmable DMA interface to the fabric switch. The node processors are based on a 64-bit MIPS core with 32KB instruction and data caches. The design intent of the SiCortex compute node is to provide for low main memory (L2 cache miss) and low communication latencies, high memory bandwidth, low power (measured in FLOPS per watt), and a complete Linux operating system support with kernel and device drivers.

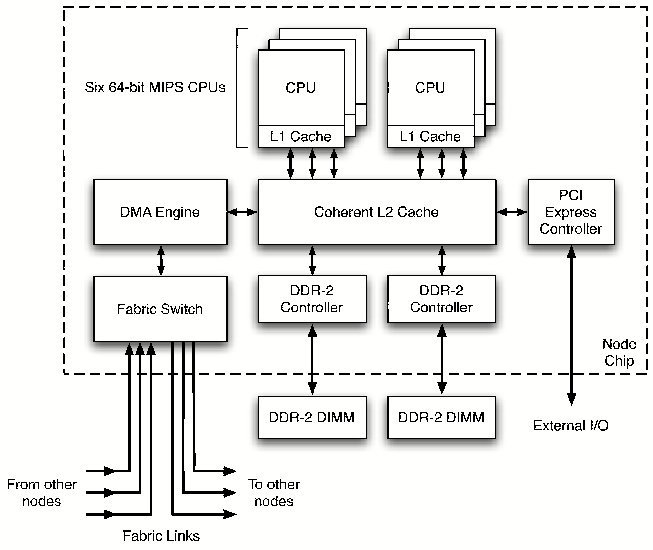

Figure 1: SiCortex Compute Node

The ultimate goal of functional verification is to prove that the design intent of the device under verification (DUV) is preserved in its implementation [2]. Thus, functional verification of the chip can be divided into two parts: design verification and hardware/software co-verification. The verification challenge is driven by the following major factors: high design complexity (198 million transistors); the presence of both the purchased design IPs and internally developed blocks; high device programmability; and Verilog simulator speed and license limitations.

2. VERIFICATION APPROACH

2.1 Verification Models

The design intent is normally stated in the design specification. A design model is needed to capture the design intent. In theory, there can be many implementations of the same design intent. The verification team created predictor and checker models from the design specification to provide for self-checking of the design and implementation intent. The architecture team developed more refined SystemC cycle-accurate High Level Models (HLMs) to capture the target architecture implementation details. The implementation team wrote Verilog RTL models based on their corresponding HLMs.

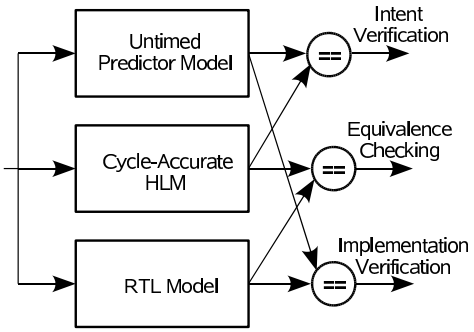

Figure 2: Verification Model Hierarchy

Figure 2 shows the verification model hierarchy that consists of three levels: intent verification, equivalence checking, and design implementation verification. The verification team started with the development of verification plans for the corresponding design blocks. The intent verification was finished upon the completion of all verification tests and the achievement of the functional coverage criteria. Then the same set of tests were applied to the RTL model. The coverage criteria in this case included both the functional coverage and Verilog code coverage data. When either functional or code coverage results were found unsatisfactory, more tests were written. For a number of blocks, such as the DMA engine and L2 cache, verification tests were run in parallel against both their cycle-accurate HLMs and the corresponding Verilog RTL implementations to prove that the HLM and RTL models were equivalent (equivalence checking) and functionally correct.

2.2 Simulation Models

The SiCortex SOC was designed from a number of purchased design IPs and custom designed blocks. All design IPs were delivered as synthesizable Verilog RTL models, whereas most of the SiCortex custom blocks had both SystemC HLM and Verilog RTL implementations. In addition, the 64-bit MIPS processor design was modelled using the SimH instruction-accurate behavioral simulator [3].

All synthesizable Verilog RTL models can be converted into fast C++/SystemC cycle-accurate models using Verilator [4]. Verilated RTL models are simulated license-free using the OSCI SystemC simulator providing more simulation cycles at no cost. Figure 3 shows the integration of various block-level simulation models intended to meet speed, accuracy, and simulation cost requirements.

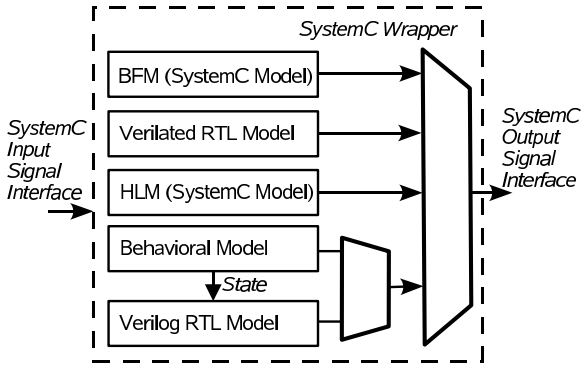

Figure 3: Integration of the Simulation Models

There are two chip simulation models: the SystemC chip model and Verilog package wrapper model. The later instantiates the RTL and gate-level Verilog chip models. Note that both wrappers are instantiated in the same SystemC testbench. In the SystemC chip model, all connectivity between the subchip blocks are described in SystemC and each C/C++, SystemC, or Verilog RTL block model is instantiated in its SystemC wrapper (see Figure 3). In addition, some blocks can be instantiated as BFMs under the control of verification test drivers that supply stimuli to and collect responses from the adjacent blocks under verification. The SOC simulation configuration, which specifies how each subchip block is modelled, is described in a Perl hash structure that instructs the build script how to compile the SOC simulation model. There are several important advantages of this simulation approach:

- It provides great flexibility for the verification of individual blocks and different combinations of subchip blocks under one SystemC chip wrapper;
- License-free simulations can be achieved by choosing any combination of Verilated RTL models, SystemC HLMs, behavioral models, and BFMs.

2.3 Testbench Design

The main goals of the testbench design are to reduce the test development cycle, facilitate the process of debugging, increase verification code reusability, and increase the level of functional coverage. The testbench implementation is based on a layered approach where each layer provides a set of services depending on the test abstraction level. There are three basic testbench layers:

- Test specification and control layer (test scenario, coverage, and test completion managers);
- Intent verification support layer (traffic manager and scoreboard);
- Design implementation verification layer (interface BFMs, predictors/checkers, and monitors).

2.4 Test Writing Methodology

Every test is described as a C++ class that inherits the ScxTest base class as follows:

```
class myTest : public ScxTest {
   virtual void init(); ///< init method
   virtual void spawned(); ///< spawned method
   virtual void final(); ///< test final method
};
```

There are three virtual methods that the test writer needs to define: `init()`, `spawned()`, and `final()`. The `init()` method is needed to reset the DUV with other verification elements. The `spawned()` method describes how to execute the test. Both the `init()` and `spawned()` methods are spawned dynamically by the SystemC `sc_spawn()` method. The launching of a test includes the instantiation of the test class and a subsequent call to the test base `run()` method shown below:

```
void ScxTest::spawnTop() {
   init();
   spawned();
}
void ScxTest::run() {
   sc_spawn_options opts;
   sc_spawn(sc_bind(&ScxTest::spawnTop, this),
            "spawnTop", &opts);
   while (!finished()) poll();
   final();
}
```

After spawning the `spawned()` method, the test enters the local while loop. Communication between a test and its BFMs is implemented via test driver methods spawned by the test `spawned()` method. The loop exits when the `finished()` method evaluates the test completion criteria as true. Finally, the `final()` method is called to collect test statistical data.

3. VERIFICATION PRODUCTIVITY

In principal, productivity can be measured by the time spent on a specific task and the costs associated with its execution. The productivity of hardware verification depends on reusability of the verification methodology and code, the use of automation tools, regression testing support, co-verification support, and control over the use of licenses [5]. Since the ultimate goal of verification is to find bugs in the most efficient way, a great deal of time was devoted to tools, code reuse, and regression testing support.

3.1 Verification Tools

3.1.1 Languages, libraries, and simulators

C++ standard template library (STL) was used throughout the project to facilitate the development of C++ verification code [6]. Also, the constraint and weighted randomization support classes and techniques provided by the SystemC verification (SCV) library were widely used [7]. To increase the verification abstraction level while handling different types of data transactions, the OSCI Transaction Level Modeling (TLM) library was used in the development of BFMs and monitors. The standard OSCI SystemC simulator was used to simulate SystemC chip models and debug tests. In addition, Cadence's Incisive Unified mixed-language NCSIM simulator was used to simulate Verilog and SystemC models.

3.1.2 Open source productivity tools

During the course of verification, Vregs and SystemPerl open source verification productivity tools were used. The Vregs tool creates Verilog headers, C++ headers, C++ classes, and verification tests for all chip status control registers (CSRs) from the specification [8]. As a result, CSR specifications and verification code are always up-to-date. SystemPerl is a preprocessor that translates simplified SystemC like code into standard C++/SystemC code for compilation [4]. SystemPerl provides a rich set of macros, acting as directives, to generate correct C++/SystemC files. SystemPerl saves close to 40% of SystemC lines, resulting in fewer typos and compile errors.

3.2 Code Reuse

Code reuse in the SiCortex verification environment was achieved primarily by developing a unified verification methodology based on a set of industry standard languages and libraries.

3.2.1 Encapsulation, inheritance, and polymorphism

C++ provides powerful capabilities, such as encapsulation, inheritance, and polymorphism, for improving code structure and reusability [6]. From the design perspective, a polymorphic base class is a base class that is designed for use by other objects. The process of creating tests requires the development of base classes with service methods designed to handle the DUV specific control and data manipulation functions. Every new test can simply inherit or encapsulate all necessary base classes to handle low level operations, whereas the test writer focuses on writing new test scenarios at a higher abstraction level. Thus, consistency, debugability, and reusability of the verification code can be maintained.

3.2.2 Verification infrastructure reuse

The real value of the verification infrastructure is in the utilization of its support layer functions and testbench elements during the development and debugging of tests [5]. All verification tests, including the testbench components, such as BFMs, monitors, checkers, and predictors, are reused to verify both the SystemC and Verilog chip models. Each test was designed using the DUV specific and common (SCV library, STL, etc.) C++ libraries.

3.2.3 Recycling subchip tests

High code reusabilty was achieved during the chip-level verification effort by reusing the tests originally written to verify subchip configurations, such as PCIe, DMA, and memory system. Below is a simplified example of the `ChipTest` chip-level test class derived from the `ScxTest` base class:

```
class ChipTest : public ScxTest {
   struct PcieBaseTest* pciTest;
   struct DmaBaseTest*  dmaTest;
   struct MemBaseTest*  memTest;
   virtual void pciSpawn() {pciTest->spawned();}
   virtual void dmaSpawn() {dmaTest->spawned();}
   virtual void memSpawn() {memTest->spawned();}
   virtual void init();
   virtual void spawned();
   virtual void final();
}
```

```
void ChipTest::spawned() {
   SC_FORK
   sc_spawn(sc_bind(&ChipTest::pciSpawn, this),
            "pcitest", &opts),
   sc_spawn(sc_bind(&ChipTest::dmaSpawn, this),
            "dmatest", &opts),
   sc_spawn(sc_bind(&ChipTest::memSpawn, this),
            "memtest", &opts),
   SC_JOIN
}
```

Note that the `ChipTest spawned()` method spawns the individual test `spawned()` methods using the SystemC fork-join construct.

3.3 Regression Testing

The value of regression testing for finding bugs is often overlooked. Random testing, where input stimuli, test parameters, and test scenarios are generated pseudo-randomly (depending on the random seed), greatly improves the verification quality by generating interesting verification scenarios. Though the majority of regression failures were not real design bugs, close to 10% of those failures can be described as either design limitations needed to be documented or interesting, hard-to-imagine test scenarios that had to be fixed in the design.

4. CO-VERIFICATION: BOOTING LINUX

The ability of the chip to boot Linux is the most important functional requirement. The software team started the debugging of the Linux kernel using the SimH behavioral standalone simulator. The total number of instructions needed to run full SMP to the user mode prompt equals approximately 16 million MIPS instructions. The Linux debug process was split into the following sequence of steps:

1. Fast behavioral simulations in the SimH environment. It takes 50 seconds to boot Linux.
2. Speed-optimized, mixed-mode, and license-free SystemC simulations (behavioral CPU models, SystemC HLMs, and Verilated RTL). The total Linux boot time is 3 hours and 27 minutes.
3. License-free SystemC simulations (verilated RTL). The Linux boot time is 14 hours and 17 minutes.
4. Verilog RTL simulations using NCSIM. The Linux boot time is 28 hours.
5. Verilog gate-level simulations using NCSIM. Booting Linux requires almost 100 hours.

5. VERIFICATION STATISTICS

The overall number of block, subchip and chip-level tests totalled almost 20,000. Every nightly regression test suite included approximately 5,000 randomly selected tests (both directed and random tests). On average, only 20% of the nightly regression simulation runs require NCSIM licenses.

The total number of critical design bugs totaled close to 1,300. Table 1 shows the distribution of critical bugs with their percentage of the total number of bugs found in HLM and RTL models of the custom built blocks. As a result, more than 80% of all bugs were found in the HLM block models and only 20% in the corresponding RTL models. The distribution of bugs is reversed at the chip-level: almost 80% of the chip-level bugs were found in the RTL chip model. A higher percentage of bugs in the RTL chip model can be explained by two reasons. Firstly, all block and subchip-level simulations are performed on the same SystemC chip model, and, secondly, the RTL chip model contains additional circuitry, such as DFT logic, PLLs, and PHYs.

Block	HLM	RTL	Total
L2 Cache	304 (90%)	34 (10%)	338
DMA Engine	217 (82%)	47 (18%)	264
FSW Switch	158 (79%)	41 (21%)	199
PCIe-PMI	159 (84%)	30 (16%)	189
CHIP	3 (21%)	11 (79%)	14

Table 1: HLM and RTL Bug Distribution

6. CONCLUSIONS

A set of fast behavioral and cycle-accurate models were developed to enable the architectural exploration, performance analysis, and software debug in early stages of the development of the SiCortex compute node architecture. Besides verifying the SOC design, it was vitally important to demonstrate that the Linux operating system and device drivers could operate seamlessly on the chip before the tapeout. The SiCortex simulation strategy provided for a higher level of control over the simulation speed, accuracy, and overall verification cost. The verification strategy and testbench design increased reusability of verification code. Open source tools, such as as Vregs, SystemPerl, and Verilator, proved to be valuable productivity tools in helping the verification team to develop, simulate, and regress tests license-free. As a result, engineers were able to run more tests and find more bugs sooner.

7. REFERENCES

[1] M. Reilly, L. Stewart, J. Leonard, D. Gingold, "SiCortex Technical Summary", 2006. (available at http://www.sicortex.com/prod_white.shtml)

[2] A. Piziali, "Functional Verification Coverage Measurement and Analysis", Kluwer Academic Publishers, 2004.

[3] R. Supnik, "Writing a Simulator for the SimH System", 2006. (available at http://simh.trailing-edge.com)

[4] W. Snyder, "Verilator and SystemPerl Environment", NASCUG, 2004.

[5] O. Petlin, A. Genusov, L. Wakeman, "Methodology and Code Reuse in the Verification of Telecommunication SOCs", 13th IEEE ASIC/SOC Conf., pp. 187-191, 2000.

[6] B. Stroustrup, "The C++ Programming Language", Addison-Wesley Professional, 2000.

[7] L. Singh, L. Drucker, N. Khan, "Advanced Verification Techniques", Springer, 2005.

[8] W. Snyder, "505 Registers or Bust", Synopsys User's Group, SNUG Boston 2001.