CIM: Component Isolation and Monitoring for System-Level Verification

Glenn Freytag and Ravi Shankar
Center for Systems Integration, Florida Atlantic University, Boca Raton, FL
{glenn, ravi}@cse.fau.edu

Tomas Tezak
Software FX, Boca Raton, FL
<e-mail address>

Abstract

Electronic system-level (ESL) design is emerging as a technique for managing the exponential growth in the complexity of system hardware and software. An ESL design flow begins with an abstract model of a complete system, which allows architectural issues to be worked out before the design is implemented in detail. However, to maximize the efficiency of this top-down design strategy, ESL simulation environments need a verification methodology that can test components of a system model in isolation without physically deconstructing the model. Such a methodology would avoid the creation of multiple testbenches to verify individual components. It would also assist in tracing simulation failures to the design components responsible for them. The methodology would ensure that each component is exercised sufficiently within the context of the system model to minimize the chance of a fault going undetected. The tests and the verification code written for the methodology should be reusable at several levels of design abstraction. In this paper, we present such a methodology based on an abstract form of boundary scan.

1. Introduction

A complex system-on-chip (SoC) presents many of the same verification challenges as a densely populated printed circuit board (PCB). A SoC design is made up of many components (both hardware and software), just as a PCB contains many devices. Ideally in both cases, a testing environment should be able to target each component individually and test it thoroughly. This means that the environment must be able to create every input condition that is possible for each component and determine the resulting output from the component. Such

comprehensive testing is not practical in the case of the PCB if the system probes only the inputs and outputs at the PCB's edge connector, because both the inputs to the PCB and the outputs from the target component must filter through many other devices. Similarly, it is not feasible to test an entire SoC design thoroughly by controlling and observing only its I/O channels.

To make individual devices more testable on a PCB, a boundary scan methodology was introduced by the Joint Test Action Group (JTAG) and was later standardized as IEEE 1149.1 [1]. The standard specifies that a boundary scan module should be inserted in every chip-level I/O path of a device. Each boundary scan module is capable of interrupting the normal data flow in or out of the device, injecting alternate input data and latching output data. In this way, all of the device's I/O activity becomes controllable and observable at the board level. A test access port is also added to the device so that the testing system can communicate with the boundary scan modules directly rather than through other logic surrounding the device. The boundary scan modules are connected in one or more serial communication chains to keep the boundary scan infrastructure simple and minimize its impact on the physical size of the device.

The 1149.1 standard, along with several derivatives and proprietary extensions, has revolutionized the testing of physical devices in finished products. However, our aim is to exploit the capabilities of boundary scan in the verification of system designs prior to production without adding code to the final product. To this end, we have abstracted the low-level boundary scan concept to define a test infrastructure for functional simulation of system-level models. The infrastructure is implemented using SystemC [2], a C++ class library that allows systems to be modeled at any of several levels of abstraction. SystemC also makes it possible to model both hardware and software in the same environment using the same set of language constructs. Our methodology is thus potentially

useful for testing either hardware or software components of an SoC design. To highlight the generic nature and adaptability of the methodology, we have named it Component Isolation and Monitoring (CIM).

2. Architecture

This section describes the various aspects of the CIM architecture, from the system-level infrastructure to the operation of individual boundary scan modules.

2.1 System model

Figure 1 shows a block diagram of our CIM architecture as applied to a system-level model. The model is assumed to consist of a top-level module in which are instantiated several modules representing the major components of the design. An Interface Module (IM) is inserted in every top-level communication path, including both the system I/O and every internal path between components. Also, three new interfaces (ctrl_in, enable_in and ctrl_out) are added to the top-level module. For a hardware design, these interfaces can be implemented as user-defined ports. For software, they could be three additional parameters of the main function.
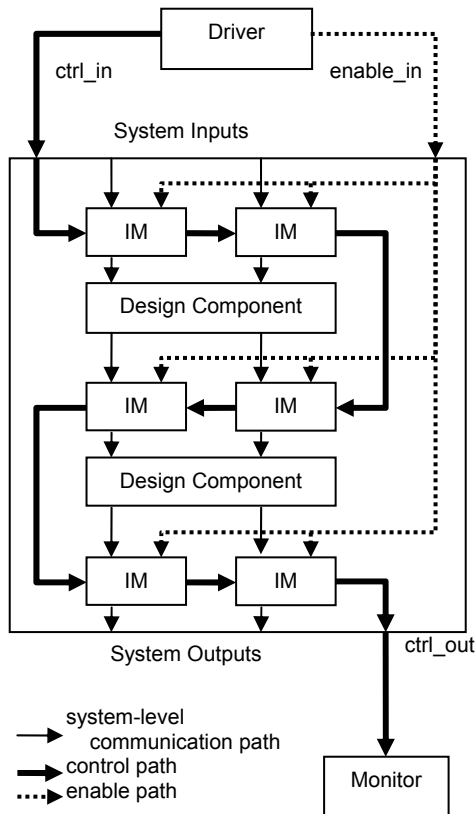

Fig. 1. CIM applied to a system model.

A control path links the IMs together in a serial fashion from ctrl_in to ctrl_out. A Driver and a Monitor use the control path to communicate with the IMs. An enable path connects enable_in to all of the IMs in parallel. The Driver uses the enable path to trigger the driving of test input and the capturing of the resulting output via the IMs.

2.2 Interface Module

The structure of the IM is shown in Figure 2. The contents of the IM may be implemented at any of several levels of abstraction, from behavioral to RTL for hardware testing or as a C++ object for software testing. The data_in and data_out interfaces are inserted into a system-level communication path as shown in Figure 1. The ctrl_in and ctrl_out interfaces are inserted in the control path, and enable_in is linked to the enable path. Each instance of the IM is assigned a unique Module Address so that the Driver can communicate with the instance individually. The Mode value determines the operating mode of the instance. The Data Pointer references a location in the simulation host's memory where a Data List for the instance is stored. The Data List can contain data that the IM captures from data_in or data that the IM drives to data_out. The Data Buffer can store either a value that arrives at data_in or a value extracted from the Data List.

The IM can be programmed to operate in any of five modes: Drive, Capture, Monitor, Pass or Isolate. Figure 3 illustrates the data flow through the IM for each mode. In each illustration, component C1 feeds its output to the data_in input of the IM, and the data_out output of the IM drives the input of C2. Each mode in the figure is described in detail below.
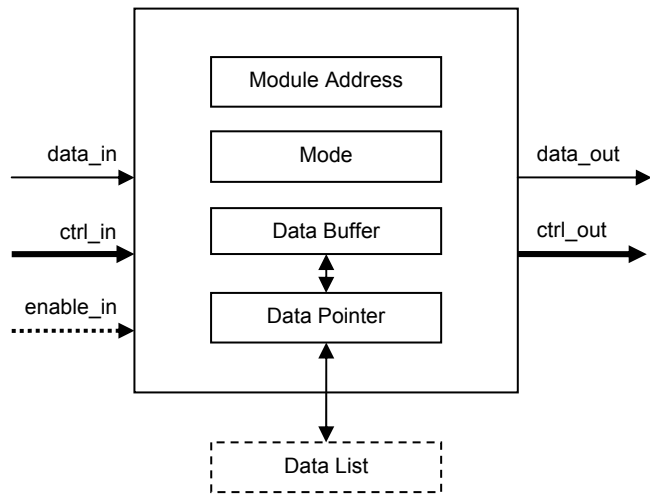

Fig. 2. Structure of Interface Module (IM).

(a) Drive　　　　　　　(b) Capture

(c) Monitor　　　　　　(d) Pass

(e) Isolate

C1　C2　components
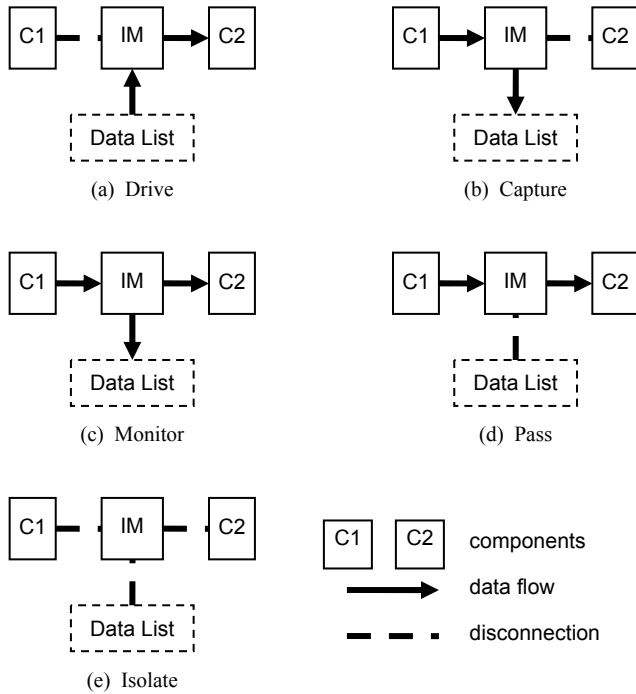
→　data flow

– – –　disconnection

Fig. 3. Interface Module operating modes.

In the Drive mode (Fig. 3a), the IM interrupts the normal data flow from C1 to C2 and instead sends input from the Data List to C2. Input from C1 is ignored. This mode isolates the input of a design component from the rest of the system model and gives the testbench direct control of the input.

The Capture mode (Fig. 3b) traps data that the IM receives from C1 and records it to the Data List. In this mode, data_out is forced to an appropriate value to simulate a disconnection of the communication path to C2. The value that represents a disconnected path depends on the data type defined for the connection (e.g. false for Boolean, zero for integer, etc.) This mode captures the output of a design component while isolating it from the rest of the system model.

The Monitor mode (Fig. 3c) records the value received from C1, but it also allows the value to pass through the IM to C2. This enables the testbench to test two or more connected design components and monitor the interface activity between them transparently.

The Pass mode (Fig. 3d) allows normal communication from C1 to C2 without recording any of the values that pass through. This mode can be used when two or more design components are being tested and there is no need to monitor an interface between them.

The Isolate mode (Fig. 3e) blocks all communication between C1 and C2. The IM should be set to this mode when neither C1 nor C2 is being tested. In that scenario, the Isolate mode saves simulation bandwidth by preventing C1 and C2 from triggering events in each other.

2.3 Packet Structure

To help simplify testing and speed up simulation, the Driver and Monitor communicate with the IMs by means of packets. Control packets are transmitted over the control path, while enable packets are sent over the enable path. The structures of these packet types are depicted in Figures 4 and 5. The Driver directs a control packet to a target IM by setting the Address field of the packet to that IM's Module Address. Each IM forwards the packet to the next IM in the control path until the packet reaches the target IM. Then, the target IM sets its Mode and Data Pointer to the values in the control packet. When the Mode and Data Pointer of each IM have been set as needed for a given test, the Driver sends out an enable packet with the Drive field set to true and the Capture field set to false. Every IM that is in Drive mode then drives input from its Data List to its data_out interface. After allowing an appropriate delay for the design to respond to the test input, the Driver sends another enable packet with the Drive field set to false and Capture set to true. Every IM that is in either Capture or Monitor mode then records the value at its data_in interface to its Data List.

2.4 Operating Sequence

During simulation, it is the responsibility of the Driver to control the CIM architecture and of the Monitor to retrieve the captured data. Before a test can be run on a system model using the CIM architecture, the Driver must initialize each IM by sending it a packet over the control path. When an IM receives a control packet, it sets its Mode and Data Pointer to the values that the packet specifies. The mode of each IM is initialized according to the location of the IM relative to the components being tested. In general, if an IM has to feed an input of a Component that is being tested, the Driver initializes the

Address	Mode	Data Pointer
(int)	(int)	(DataList *)

Fig. 4. Structure of control packet.

Drive	Capture
(bool)	(bool)

Fig. 5. Structure of enable packet.

IM to Drive mode. If an IM must record the output of a Component that is being tested, the Driver initializes the IM to Capture mode. If an IM lies in the path between two or more Components that are being tested together, the IM may be initialized to either the Pass or Monitor mode. All other IMs should be initialized to the Isolate mode to disable all Components that are not being tested. The Driver must also program the Data Lists for all IMs that have been initialized to Drive mode.

Following initialization, the Driver sends a packet over the enable path to trigger a drive operation in every IM that was set to Drive mode. Each of these IMs then drives the first input value from its Data List to its data_out interface. After allowing an appropriate delay for the design to respond to the test input, the Driver sends a packet over the enable path to trigger a capture operation in every IM that was set to either Capture or Monitor mode. These IMs then record the values at their data_in interfaces to their Data Lists. The Driver triggers another drive operation for the next value in the Data List of each driving IM and a corresponding capture operation in all of the capturing or monitoring IMs. The drive/capture sequence is repeated until the Data Lists of all the driving IMs have been consumed. Finally, the Monitor reads and reports the contents of the Data Lists where captured data was stored.

3. Design examples

In order to study the feasibility of using CIM as a method for verification of SoC designs, we selected two design examples and deployed our Interface Modules around them.

3.1 Four-bit adder

We chose a four-bit adder for our first application of CIM because of its simplicity and easy adaptability to different levels of design abstraction. The adder module is depicted in Figure 6. All of the ports on the module represent individual bits rather than bit vectors. The adder therefore has nine inputs, four for operand A, four for operand B and one for the carry in. Similarly, there are five outputs, four for the sum and one for the carry out. Thus, the adder requires fourteen IMs to test it.

One important issue to address in developing the CIM architecture is its scalability in terms of code length and
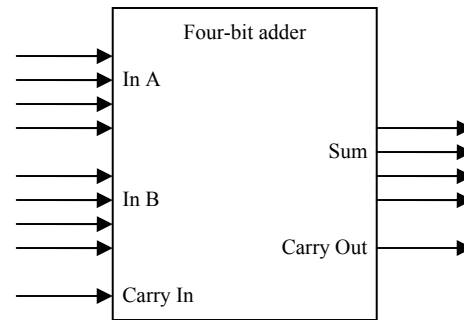

Figure 6. Four-bit adder module.

simulation speed with increasing design complexity. To explore this issue without having to use multiple designs, we created several versions of the four-bit adder at different levels of abstraction. The initial implementation of the architecture was done around an adder modeled at a behavioral level with the C++ addition operator. We then created a Boolean version where we assembled the four-bit adder out of half adders and full adders. The operation of each half adder and full adder was defined in terms of logic equations. Finally, we created a gate-level model of the four-bit adder where we implemented the logic equations of the Boolean version using instances of a NAND-gate module. The three different models, though they were different in their internal structure, had the same I/O ports and worked with the same data types. Therefore, our testbench for the four-bit adder was directly compatible with all three models.

Table 1 shows the effects of CIM on our design examples in terms of lines of code and simulation time. The addition of CIM to our three adder models increased the line count by 430 in each case. While this almost doubled the code size of our behavioral adder, the effect on the Boolean and gate-level versions was somewhat less pronounced. The behavioral adder increased the most in code size because it was the simplest of the three adders to begin with. The simulation time for each type of adder is the total time required to test all 512 possible combinations of input values. Again, the behavioral adder showed the largest increase because of the simplicity of the original adder code. However, the Boolean version had the least increase in simulation time, even though the gate-level adder had the highest original time.

Table 1. Effects of CIM on design examples.

Design example	Lines of code		Simulation time (sec)	
	Original	CIM	Original	CIM
Adder/ behavioral	445	875 (+97%)	0.106	0.159 (+50%)
Adder/ Boolean	531	961 (+81%)	0.1402	0.162 (+15%)
Adder/ gate-level	612	1042 (+70%)	0.203	0.24 (+18%)
Robot controller				

3.2 Robot controller

Next, we applied CIM to a SystemC model of a simple robot controller. The model consisted of a controller module, a read-only memory which programs the controller, and a monitor which receives signals from the controller and provides feedback. We deployed a total of 23 Interface Modules along the communication paths between the three components of the model. Some of the Interface Modules were bidirectional versions of the IM described above, enabling us to drive or capture test data through either data port of the IM.

4. Discussion

Although the four-bit adder example is admittedly quite simple, it helped us confirm what we believe to be the advantages of the CIM methodology. Our CIM-based testbench was directly compatible with adders modeled at different levels of abstraction, attesting to the reusability of the CIM architecture. The addition of CIM to our test environment increased the total code size by a constant number of lines, regardless of the level of detail in the design itself. CIM should therefore have a relatively small impact on the code size of designs that contain more internal complexity. The simulation overhead of CIM may seem rather high, but this is only because of the simplicity of the test case itself. The relative overhead was much less for our Boolean and gate-level adders than for behavioral, suggesting that it could be even lower for a more complex design.

It should be noted that our use of the same design model at different levels of abstraction was only intended to test the methodology with varying design complexities. Our methodology remains targeted at system-level designs, and our ultimate goal is to create a tool that can automatically modify such designs to insert the verification architecture as described in this work.

Automation of our methodology should help to maximize its efficiency in all respects.

5. Conclusions

The CIM methodology has the potential to enable more thorough testing of SoC models in the design phase. CIM allows various parts of a system model to be tested in isolation from the rest of the model. The methodology is compatible with designs at several levels of abstraction, thus providing a reusable testbench structure. The CIM architecture itself is conceptually straightforward and is modeled at a high level of abstraction, which simplifies is operation and reduces its overhead.

6. References

[1] *IEEE Standard Test Access Port and Boundary-Scan Architecture*, Std. 1149.1, 2001

[2] T. Grotker et al, *SystemC for System Design*, Kluwer Academic Publishers, Boston, 2002.