SOC Verification Software - Test Operating System

Robert Devins
IBM Corp. Microelectronics Div. M.S 862J, 1000 River St. Essex Junction, VT 05452
rdevins@us.ibm.com

Abstract -- *The IBM Microelectronics World Wide Design Center (WWDC) provides System-on-Chip solutions to its customers. SOC's are typically built from IBM and customer cores, often containing an embedded processor. The SOC business is competitiveand time-to-market pressures are increasing. The chips are becoming more complex - driving enhancements in system programming and verification techniques, reusability, and simulation environments.*

This paper discusses techniques that the WWDC is using in system verification software development. It outlines a strategy of providing reusable test programs for verification utilizing customer reusable low-level device drivers and simulation speedup techniques that enable effective system level verification and coverage. The goal is to dramatically reduce system-on-chip development time and cost, while improving overall quality.

I. Introduction

System-on-Chip development requires a unique perspective on verification and programming practices, and opens the door to a new paradigm in the approaches to be used. The explosion in SOC business demands reduced time and manpower for each chip project, and the increasing complexity of the chips must be addressed with a comprehensive verification environment. SOC's are not just hardware - the software running in the chip, or software using the chip, ultimately comprises the functionality of the product. Indeed, high-function, generic SOC's are being redefined in a multitude of products simply by installing different software. A key element to meeting SOC development objectives is reuse. Reuse not only applies to hardware IP, but to verification programs, crossing into diagnostics and system software.

The method that IBM's WWDC has chosen to meet these objectives is a software program called the Test Operating System (TOS)[1]. The primary goal of TOS is to provide a platform for developing reusable system test programs, performing scenario-based verification of a complex system, and controlling the simulation environment. TOS-based verification is capable of emulating real-life usage of the SOC hardware by exercising concurrent core functionality, as well as determining that all components are connected properly. Because TOS is a software program, it is further capable of providing initial hardware bringup diagnostics.

The software-based verification environment helps bridge the gap between the silicon provider (generating hardware) and the customers (generating software). TOS source code is readily available to customers to aid in generating the customer "value add" software, especially in device drivers. Customers are very happy to get source code as a starting point; the WWDC is winning business because of this.

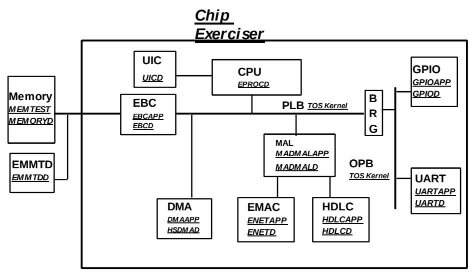

Figure 1: Hardware/Software Built Together

To maximize reuse, the TOS method enables development of verification programs, called Test Applications, and example device drivers called Low Level Device Drivers, to be associated with each hardware component. This allows system designers to stitch together a verification environment at the same time, and in the same manner as the hardware chip is stitched. There exists a library of hardware IP (cores and processors), and a corresponding library of verification IP (apps and drivers). When a piece of IP is chosen, it comes complete with verification software. The TOS kernel enables plug-and-play inclusion of code very easily so software integration is done in a very short time, often ahead of hardware integration.

Apps and drivers are written to verify a core, independently of a potential chip that the core may be used in (similar to the way the reusable hardware IP is designed) and are programmable, via parameters, to enable flexibility. This is the essence of reuse. Therefore, a method of linking apps and drivers in a specific chip is required. In TOS, a top-level exerciser program is used to bind together individual test programs, and to assure co-operability of the programs in a concurrent system environment.

One of the key elements of the TOS is its ability to multitask test applications. This enables hardware contention and concurrency exercising, which is vitally important to real-life functional emulation. Multitasking also serves to improve simulation throughput. Parallel test execution allows many testcases to be run in a single simulation run.

2. Software Structure

TOS is specifically designed for system level verification. Such an environment poses certain restrictions on a system, particularly the speed of today's simulators are not conducive to complex software execution. TOS contains the necessary elements to meet its objectives, and is expandable to future simulation and acceleration technologies as they become available. The modular design and consistent message-based interfaces enable easy addition of test applications and low level device drivers into a system. The design of the message interfaces allows multi-tasking to occur between applications, thus allowing hardware functional overlap and contention to occur, while individual applications are not aware of such overlap. Concurrent execution requires the kernel to provide support for resource allocation such as a memory allocation manager and mutual exclusions (mutexes). All code is written in "C", except for a special device driver which directly supports the embedded processor, or a virtualization of a processor. It is in this driver, and only this driver where assembly language may be used. This modularity allows TOS to support systems with different processors, systems with no processors, and native workstation execution. The TOS kernel provides a generalized, processor independent interrupt callback mechanism, and a consistent set of functions that allow chip and processor specific initialization sequences.

NonCooperative vs Cooperative Execution

Not all testcases can be run concurrently with other tasks. For example, a fundamental register test being done on a common core, such as an interrupt controller, can cause failures in dependent applications. Such a test is considered "uncooperative". TOS executes tests in two groups, with the uncooperative tests running **serially** at the beginning of a run, followed by the cooperative tests running in **parallel**. Some tests designate themselves to the system as strictly uncooperative, and in some cases, it is up to the system verifier to select the appropriate classification of tests for a specific chip implementation.

Multiple-Instantiation of Apps and Drivers

It is a requirement for reusability that code designed to program and verify a hardware component be plug-and-play. It is feasible (and practical) that more than one copy of a core be accommodated by the system. Some cores are multi-channel, and can run simultaneous testcases on each channel, this also must be accommodated. TOS has constructs that enable all apps and drivers to be written so they can be used, simultaneously, as many times as required in a design.

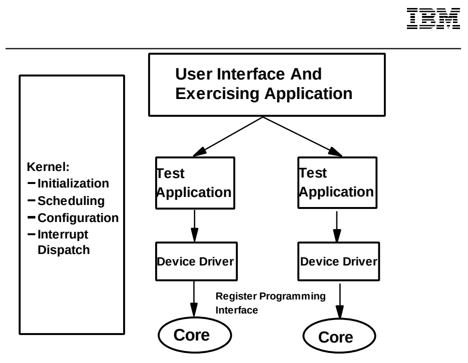

Figure 2: TOS Software Structure

TOS is structured similar to many common operating systems. All operation is controlled by a scheduling kernel, which is responsible for dispatching and managing the tasks. In this paper, the terms "tasks", "test application" and "application" are used interchangeably. The kernel contains the resource and memory management functions, the processor initialization and first level interrupt handling functions. Chip specific code is included in the "exercising" application program, designated as the "master" task. The master task is by default always scheduled to run, has decision control over what to run, and when to run it. Specific chip initialization, testcase selection and specific testcase setup is perform when running the master task. Once selection and setup is complete, test application tasks are scheduled in the kernel via the message interface.

Hierarchically, chip specific knowledge is in the exerciser, and core specific knowledge is in the device drivers. Drivers know how to program the function of the underlying core hardware, but perform no verification algorithms, they are service routines for the test applications. Test applications contain the specific knowledge and algorithms to perform tests using the services of the drivers, then verify the results. All test applications are self-checking. The exerciser contains functions for selecting meaningful application tests (for the given chip), initializing and scheduling them in the kernel.
All exercisers and test applications follow a similar control flow. Each test application supports two chip-specific functions that are linked into the system exerciser: a **starting_function** and an **ending_function**. The starting functions are used to initialize (obtain resources, setup chip specific logic) and select testcases that are available in the test application and are appropriate to run in the system. Starting functions are called from the master task whenever the application is idle. Ending functions are used to free resources and are called whenever the application has just completed a test. Testcase results are logged by the ending functions.

Test applications are written to be multi-tasking. All applications are invoked from a message. (Testcases are **messages** to TOS test applications) If the application has never been run, it is given the opportunity to perform one-time initialization code, then it begins a forever loop, where it waits for a message. Since the initial invocation was a message, the application will immediately wakeup and start processing. When done, it will again wait for a message. An application is considered **idle** if it has never been run or is currently waiting for a message.

TOS is a non-preemptive operating system, all applications must be good neighbors, and yield control periodically. Since most testcases involve some sort of hardware data transfer, applications will issue the kernel **YIELD** directive after hardware is started, during the wait for completion. Applications that are yielding stay on the schedule and are continuously called (and are said to be in a **yield-loop**).

The application remains in the yield-loop until a countdown timeout or hardware completion is detected (via an interrupt handler callback or hardware poll). At that time, the application executes a results verification algorithm, sets a success/fail return code and waits for the next message.

Interrupts are handled asynchronously relative to applications. Device drivers initially service hardware interrupts, determine and remove the cause and issue callbacks to appropriate applications to signal interrupt events. An event handler, or **callback function** in an application sets an internal flag which is monitored by the **yield-loop**. When the flag is set, hardware is determined to have completed it's operation.

Typical YIELD-LOOP Construction

```
timeout=APPLICATION_TIMEOUT_COUNT;
transfer_complete=0;
while(timeout-- && !transfer_complete)
{
  YIELD;
  Get_Instance_Context();
}
```

The flag "transfer_complete" is set asynchronously by the application callback function. This is normally the terminating condition of the loop. The YIELD statement returns control to the kernel where other applications are dispatched, which may include another instance of the current application. When dispatched, the application code returns from the YIELD. Since this application may be running in multiple instances, the proper instance context must be reestablished in all cases of kernel dispatch.

4. Testbench Structure

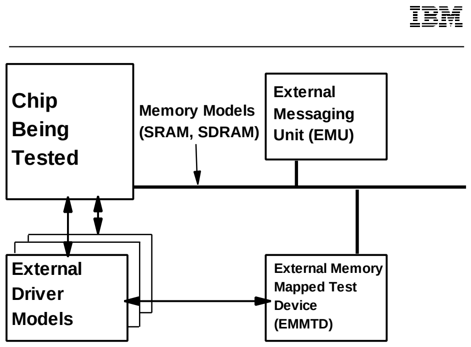
IBM
Chip Being Tested
Memory Models (SRAM, SDRAM)
External Messaging Unit (EMU)
External Driver Models
External Memory Mapped Test Device (EMMTD)

Figure 3: Components of Testbench

System verification often involves external, or off-chip data transfer or control. TOS environments consider the off-chip platform as a TOS **testbench**. Testcases must be able to send and receive data according to protocols as a function of verifying chip IO connections and system functionality. Typically, this is done using external stimulus/expect models. Models come in many varieties, from simple wrap-back connections to complex vendor protocol drivers. Because TOS is a software-based environment, it is necessary for code inside the SOC to be able to control and monitor the activities of external devices.

TOS software uses a memory-mapped external device (EMMTD - external memory mapped test device) to communicate with external behavioral devices. EMMTD is simply a bi-directional general-purpose-IO device containing banks of registers (currently 128, 8-bit registers), connected to one of the SOC's external memory busses. Software reads and writes the registers, which appear as wires to trigger or get responses from an external model.

The testbench also typically includes a memory-mapped write-only device for communicating (tracing) internal software execution to a simulator or user console. The EMU (external messaging unit) simply contains trace decoding logic that converts register writes into human readable display messages. This function has proved very useful for realtime execution tracing and debug. The EMU timestamps all trace entries so information can be cross-referenced to a simulation waveform display.

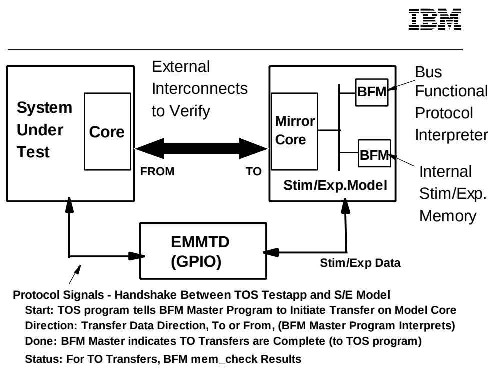
IBM
External Interconnects to Verify
System Under Test
Core
FROM
TO
Mirror Core
BFM
BFM
Stim/Exp.Model
Bus Functional Protocol Interpreter
Internal Stim/Exp. Memory
EMMTD (GPIO)
Stim/Exp Data
Protocol Signals - Handshake Between TOS Testapp and S/E Model
Start: TOS program tells BFM Master Program to Initiate Transfer on Model Core
Direction: Transfer Data Direction, To or From, (BFM Master Program Interprets)
Done: BFM Master indicates TO Transfers are Complete (to TOS program)
Status: For TO Transfers, BFM mem_check Results

Figure 4: Mirror Core Model Example

A simple method of developing a complex model is to mirror the internal core on the testbench. The EMMTD device serves the trigger/response handshake to the mirror core subsystem. The mirror subsystem contains simple handshake interpretation logic required to program the mirror core under the direction of SOC test software.

This example shows how the EMMTD is used to control a mirror using four interface signals. After the internal core is programmed by software, **Start** is activated to trigger the mirror to configure its core to either send or receive data (according to the **Direction** signal). When configuration is complete, **Done** is asserted by the mirror, and SOC code begins the transfer. If data is going to the mirror, it is verified in the mirror subsystem logic. Data going to the SOC is verified in the TOS software. To determine the success or failure of mirror verification, SOC software interprets the **Status** signal.

EMMTD is used in a variety of ways, depending on the needs of the test applications. Anything from single wire triggers to complex busses may be implemented.

4. Simulation Execution Modes

TOS software is designed to seamlessly operate in a variety of modes to accommodate the requirements of simulation users. Event-driven simulators are not conducive to running complex software programs completely in the confines of the HDL simulator model. Nor is software debug made easy in these environments. Techniques are needed to improve the performance of the system and its debuggability.

RTX - Simulator Independence

When simulating a design in one of the modes described in this section, it must be noted that the TOS software, the ISS and the simulator itself are run under the control of an overseer program called RTX (run time executive)[2]. RTX is responsible for providing a simulator independent interface to allow TOS verification to be seamlessly run on a multitude of different kinds and brands of simulators. All interprocess communication (IPC) is managed in the IPC layer of the RTX control program.

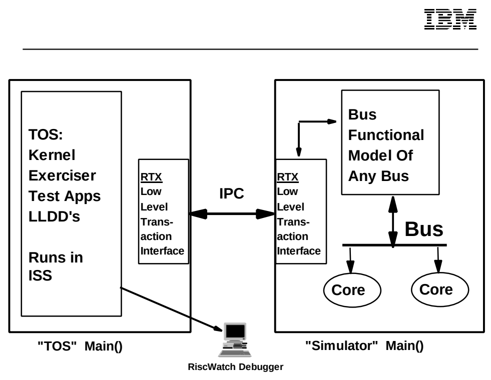
IBM
TOS: Kernel Exerciser Test Apps LLDD's
Runs in ISS
RTX Low Level Trans-action Interface
IPC
RTX Low Level Trans-action Interface
Bus Functional Model Of Any Bus
Bus
Core
Core
"TOS" Main()
"Simulator" Main()
RiscWatch Debugger

Figure 5: ISS/BFM Execution Mode

A common enabler for pre-silicon software developers is the Instruction Set Simulator (ISS) model. The ISS is a workstation-based C program that interprets and executes specific processor instructions. ISS models often come with an associated source code debugger that allows single-step and breakpoint capability. When coupled with a logic simulator, a co-simulation system is formed. In this case, the embedded CPU model is removed from the HDL simulation, and replaced by a bus-functional model (BFM). The BFM is capable of driving and monitoring the appropriate CPU bus signals on behalf of the ISS.

The ISS method is used in IBM's WWDC department. All TOS software is executed in a PowerPC(TM) ISS, running in a workstation process. Bus transactions are issued across the IPC to another process containing the HDL simulator, where they are driven by the BFM. The ISS process is stalled until the bus transaction is completed. Users develop and debug TOS sofware in this mode using the PowerPC-Based RISCWatch(TM) source level debugger. This mode offers a reasonably responsive simulation environment for testcase software developers, but has a serious drawback: The SOC model does not contain a true processor, and is not executing any code. Verification regressions, or gate-level regressions cannot be run on such a simplified model.

To allow reasonable throughput for full-function regression simulations, the split-domain execution model was developed. Split domain is designed for verification regression runs, it is not conducive to software development.

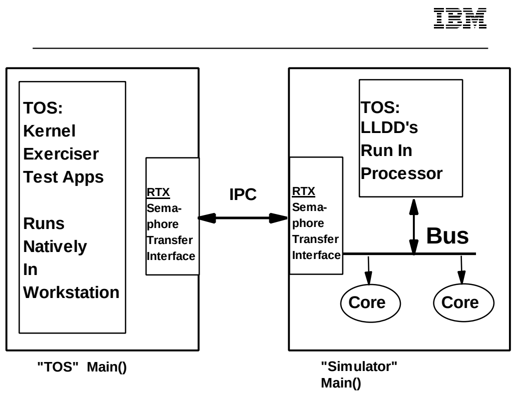
IBM
TOS: Kernel Exerciser Test Apps
Runs Natively In Workstation
RTX Sema-phore Transfer Interface
IPC
RTX Sema-phore Transfer Interface
TOS: LLDD's Run In Processor
Bus
Core
Core
"TOS" Main()
"Simulator" Main()

Figure 6: Split-Domain Execution Mode

In split domain mode, the TOS software is split into a high-level domain containing the TOS kernel, exercisers and test applications. The high-level domain runs in a process separate from the simulation process. All low-level device drivers reside in simulation memory and are executed by the CPU-based simulation model. This allows much (up to 80%) of software overhead to be executed at workstation speeds, while still executing real CPU cycles in the simulation model. High-level code executes until a device driver is called (unlike BFM mode where a bus cycle is required), and control is transferred to the simulator to execute the device driver function in CPU software.

TOS execution modes are transparent to software developers; the differences are completely encapsulated in the kernel API's.

5. Developers Tools

As the WWDC grows and takes on more business, it will become necessary to make the TOS system available to more internal user groups, contractors and customers. To make the transition easier on both the TOS users and TOS developers, a set of GUI-based tools, called the Autotos project, has been developed. The GUI will enable user access to all future functionality of the system.

As is true with many software systems, an integrated developers environment is available for TOS users. The primary focus of the tool is ease of use, and development efficiency. Autotos combines the use of code generation wizards and software component stitching with an integrated Graphical User Interface. This allows users to refer to the GUI to do everything from developing test application/driver code, constructing a system and test scenarios to launching a simulation run and monitoring results.

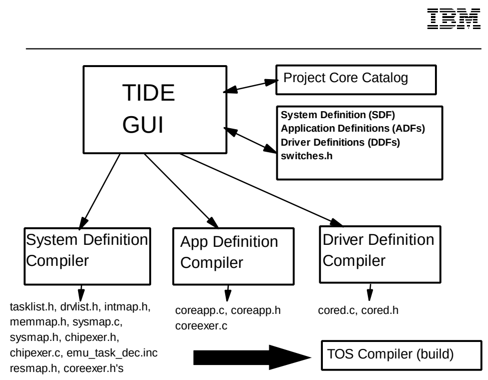
IBM
TIDE GUI
Project Core Catalog
System Definition (SDF)
Application Definitions (ADFs)
Driver Definitions (DDFs)
switches.h
System Definition Compiler
App Definition Compiler
Driver Definition Compiler
tasklist.h, drvlist.h, intmap.h, memmap.h, sysmap.c, sysmap.h, chipexer.h, chipexer.c, emu_task_dec.inc resmap.h, coreexer.h's
coreapp.c, coreapp.h coreexer.c
cored.c, cored.h
TOS Compiler (build)

Figure 7: Autotos Architecture

Autotos is composed of three primary components linked together by the GUI. Users will work within specific projects, and select TOS components from a common catalog (library). The system-definition section enables system designers to selected from the catalog and configure the system as required for the project. Test scenarios and options are specified and simulation sessions are launched. The application and driver sections are used by test application and driver developers to create or modify the TOS applications and drivers. These tools are code generation wizards which automatically create tailored TOS templates and allow developers to focus on app/driver code rather than TOS interface code. Code generation is not used only once to create templates, it is iterative - tests and interfaces are added, modified and deleted. The tools perform read-modify-write operations on source code according to the user directives.

The autotos GUI is expected to be run as a Web Based application. This enables engineers working at customer sites, teachers holding remote classes, or contractors to gain access to the latest catalogs and tools online. Subject to contractual and local law restrictions, simulations and compilations can be performed remotely, further enabling external or IBM field groups access to the TOS technology.

6. Summary

The exploding System-On-Chip business coupled with the need to reduce cycle time and produce high quality complex devices has led to the need for a reusable system verification methodology. SOC customers are often software suppliers looking to bundle their "value add" into IBM provided chips. The TOS methodology helps bridge the gap between silicon suppliers and software-based customers, by providing customers with examples of working software, and by performing real-life scenarios on designs prior to releasing them. As more people use the methodology, tools must be provided to assure efficiency and ease of use. IBM is winning customers because the TOS verification system and its related tools is an available technology.

Acknowledgments

The author gratefully acknowledges the contributions of the IBM WWDC teams for the creation and use of this methodology. In particular, Ken Mahler for advice and implementation, Mark Kautzman for project management, Bob Herzl, Jim Robinson, Dave Milton and Joe Williams for advice and counsel.

References

[1] R. Devins "System On Chip Verification and Programming". *IBM MicroNews*, vol 5, no 2, pp. 22-23, April 1999.

[2] K. Mahler, "An Object-Oriented Simulator Independent Environment for Logic Verification of Systems on a Chip", DesignCon '99 February, 1999.