FPGA prototyping technology has been widely used in microprocessor design and verification.\textsuperscript{1,2} Running massive tests on an FPGA prototype can cover boundary conditions that software simulation typically misses. Moreover, the at-speed feature of such platforms makes it possible to provide the huge amount of test vectors required by the complex processor functionality. System-level functions (SLFs) such as memory management and interrupt handling are essential parts of modern microprocessor architectures. However, verification of SLFs on FPGAs is becoming increasingly difficult because of SLFs’ inherited close interaction with software such as the operating system. A framework that efficiently builds meaningful software scenarios on a hardware prototype is a key requirement to cope with such challenges.

However, we are not aware of any significant effort to develop such a framework on FPGAs for verifying SLFs. Recent random-test generators, though successful in software simulation, cannot be used with hardware platforms, because they rely on the behavioral test-bench to build software contexts. Operating systems, on the other hand, though widely used as test vectors on FPGAs, are inefficient for verification because of the huge debug overhead induced and the limited control provided.

This article proposes the Verification-Purpose Operating System (VPOS), a flexible framework for SLF verification on FPGA prototypes. Inspired by the concept of the general-purpose operating system (GPOS), VPOS is a purely software-based, platform-independent test scenario manager that can be seamlessly integrated with existing test generation mechanisms. It handles the initialization of machine resources and reproduces different software scenarios for both hand-written and automatically generated tests, making the verification process as realistic as running a GPOS. But, unlike the latter, VPOS is specifically designed for hardware verification, so it can be flexibly configured to verify a particular SLF with high coverage. In addition, our VPOS-based methodology reduces verification time by running massive, long, random tests on a high-performance FPGA and rerunning failed tests on a flexible software simulator. (The “Related Work on SLF Verification” sidebar discusses various other approaches to SLF verification.)

How VPOS builds software contexts

Figure 1 shows the conceptual block diagram of VPOS-based verification on an FPGA platform. The compiled VPOS and tests are loaded into a flash memory. On power-up, VPOS first copies itself from flash to memory. Then, it repeatedly loads different tests from flash memory, sets up the processor resources and software contexts, and executes the test cases.
VPOS algorithms

VPOS is responsible for case scheduling, memory management, and interrupt handling for test cases. However, simplified algorithms are used instead of GPOS-based ones. These algorithms include static case scheduling (SCS), static memory management (SMM), and pseudo-interrupt handling (PIH). Other GPOS features such as interprocess communication,
file systems, and I/O control are not necessary for verification, so they aren’t included in VPOS. We also introduce a data structure called the case control block (CCB), so that verification engineers can flexibly control the behavior of VPOS and run tests on it.

**Case control block.** Test cases play the same role in VPOS as tasks play in a GPOS. The latter usually includes a task control block (TCB) to run user applications (i.e., tasks). We adopted this concept for VPOS by designing a corresponding CCB to enable the VPOS to build software contexts for tests.

For each test case, VPOS maintains one CCB, which contains all the information necessary to build software contexts. Figure 2 shows an example of a CCB implemented using assembly language and comprising the following sections:

- **CCB length (line 1).** This is the length in bytes of the current CCB.
- **Memory allocation address (lines 2-5).** This section records the starting and ending addresses of the case in flash memory, and the corresponding target address in memory—that is, the map from the flash-memory address to the physical-memory address. In Figure 2, the test case’s code and data sections are mapped from address 0xfff20000 and 0xfff40000 to 0x20000 and 0x80000, respectively.
- **Scenario fabrication (lines 6-7).** This section records the machine state register (MSR) value and the test program entry, which the CCB uses to support test case scheduling in VPOS. The example in Figure 2 enables data address translation (MSR_D) and user mode execution (MSR_USR) in the machine state register, and sets the case entry to 0x20000.
- **MMU register configure (lines 8-12).** The MMU register configure includes the MMU register value and the page table entry (PTE) definition—that is, the map from the effective address (EA) to the physical address (PA). The PTE in our example enables address translation from 0x40000 to 0x80000 and sets memory protection to Supervisor_Only.

The details of the CCB data structure can be highly architecture dependent, but here we present just the general information for this structure.

CCBs of all tests are collected and organized into a CCB table (CCBT) data structure, as shown in Figure 3. The CCBT contains an array of pointers to entries of each test’s CCB. By reading the pointers, VPOS builds software contexts according to the data in the CCB of a particular test and then starts the test.

**Static case scheduling and memory management.**

A GPOS dynamically performs task scheduling and memory management to maximize software performance. However, to focus on hardware verification and enable highly flexible user control over the software contexts of tests, VPOS uses static algorithms.

![Figure 1. Verification-Purpose Operating System (VPOS)-based verification on an FPGA platform. (DUT: device under test.)](image)

![Figure 2. Example of a case control block (CCB).](image)
Here, “static” means deciding case scheduling and memory management when the test case is developed rather than at runtime. By setting each section in a CCB before running tests, verification engineers can easily combine the assorted MMU and scheduling scenarios to verify SLFs.

Memory management can be described as two address maps. The first map tells the operating system how to allocate and copy the program from flash memory to a PA. The second map is the address translation between the EA and the PA. In VPOS, a CCB statically and explicitly defines these two maps.

VPOS simplifies the task-scheduling algorithm in a GPOS to control test cases. In a GPOS, scheduling merely involves fabricating an interrupt scenario (machine state and task entry point) dynamically. However, VPOS performs this statically because the test case’s scenario fabrication information is in the CCB. VPOS copies those values from the CCB to special interrupt registers, and the processor execution flow changes from the fabricated interrupt scenario created by VPOS to the given test case entry point with the given machine state. The test case, however, starts execution as if it returns from a real interrupt without noticing the existence of VPOS.

**Interrupt handling.** VPOS discards the complex, software/hardware-interactive interrupt-handling mechanism in normal systems and uses pseudo-interrupt handling to check hardware features directly. The interrupt service routine (ISR) doesn’t provide traditional services; instead, it checks interrupt-related register settings.

The handling process is straightforward. The main test case puts the expected value of all interrupt-related register settings in general-purpose registers (GPRs) and executes the offending instruction. If the interrupt doesn’t occur as expected, the main test executes a consequent instruction (a trap), which branches unconditionally to error handling.

If the interrupt occurs as expected, the test case proceeds to the pseudo-interrupt handler. Figure 4 shows an example handler of a memory protection violation interrupt. Usually, in this kind of interrupt, hardware places the program counter (PC) of the interrupted instruction and the EA of the offending memory space into special-purpose registers (SPRs). The example handler first checks the two register settings with the expected value in the GPRs (lines 1-4). Then, it returns to the main test case by setting a new return address to the PC, to avoid entering the trap (line 6).

**Implementing VPOS**

In our project, VPOS is implemented in a combination of assembly language and C. It consists of five main parts:

- **VPOS kernel.** This is the code to perform SCS and SMM. It repeatedly reads the CCBT and builds software contexts for tests.
- **Interrupt handler.** Some interrupts in VPOS use PIH, and others use real handling routines, depending on the purpose of the current test.
Verification of System-Level Functions in Microprocessors

- **CBT.** The CBT resides in the data section of VPOS. Each pointer is initialized at boot time, and each actual CBT resides in flash memory along with the test case’s main code.

- **Bootloader section.** On power-up, VPOS first must copy itself to memory, which is a read-and-copy process. It then initializes the CBT to locate each CCB.

- **Debug routine.** This routine includes a mini UART (universal asynchronous receiver/transmitter) driver that communicates with the engineers and reports the results of individual tests.

### Verification process using VPOS

We used VPOS for FPGA verification of the SLFs of a 32-bit, dual-issue RISC microprocessor designed by our institute. This processor supports independent instruction and data memory management, as well as precise interrupt handling. The methodology suggested here builds a consistent test generation and result-checking mechanism between software simulation and FPGA verification so that we can switch seamlessly between them and make them closely interact with each other to maximize verification efficiency.

**Software simulation environment**

In this stage, a test generator is created to provide stimuli. This test generator reads the test templates written by the verification engineers, and generates main assembly test programs and their corresponding scenario description files, which are used to inform the testbench about how to build software contexts for SLF verification (see Figure 5a). The random generation of tests and their software contexts are not novel ideas for software simulation. But they are not the main focus of this article, so we don’t discuss them in detail here.

In our project, we implemented the test generator using Perl. The testbench, therefore, initializes the DUT resource according to the scenario files, runs the generated assembly tests, and compares the DUT response with the built-in reference ISS (inline comparison).

**FPGA verification environment**

During FPGA verification, long, directed, or fully random tests are generated instead of short ones to enable massive, exhaustive running on the FPGA for boundary conditions. To do this, test template writers...
should specify as few constraints as possible. The test generator and the reference ISS in software simulation are reused with two small modifications: self-checking tests and CCB generation.

Regarding the first modification, unlike the inline comparison technique in software simulation, there is no reference ISS on the FPGA, so the tests should be generated in a self-checking manner. Upon completing a specific functional point, the test calls a checking routine that computes the signature of the result by performing an XOR on all register values, and compares this signature with the expected signature in the memory. The expected signature is obtained by prereunning the test on the reference ISS before FPGA verification begins, and is compiled as part of the data section (see Figure 5b) to form a full self-checking test. In other words, a self-checking test extends the main test code with a checking routine in the code section, and with an expected signature in the data section. This method can be regarded as an offline comparison between the reference ISS and the DUT. The reference ISS is moved out of the behavior testbench and is integrated as a part of the test generator to compute the signature for each test.

Regarding the second modification, in an FPGA environment, scenario files describing software contexts for the main tests are no longer capable of running. Instead, they are converted and gathered by a CCB generator into the CCBT. The latter is then combined and compiled with VPOS to form the runnable firmware that controls all tests (see Figure 5b). The CCB generator reads the input scenario file; maps it, line by line, to CCB format; and writes the CCB into the data section of VPOS. In our project, this CCB generator is a source-to-source compiler implemented using Perl.

Verification engineers can use the EDA tool to download tests and VPOS into the FPGA board’s flash memory through the JTAG port of the FPGA. In our project, we use the Stratix II EP2S60 FPGA development board and Quartus II 7.1 software to build the platform. The test generator and the CCB generator are called from the command line script of SOPC Builder 7.1 (see Figure 6).
Debugging the failed tests

The tests are self-checking. Thus, it’s up to each individual test to check whether it passes or fails and to record its ID. VPOS, however, informs the verification engineers of the test result and the test ID through a UART port.

Each failed test must be rerun on the software simulator (see Figure 6), because the test generator and reference ISS for software simulation and FPGA verification are inherently the same. By feeding the test template to the test generator for software simulation, the engineers can easily reconstruct the non-self-checking version (i.e., the inline-compare version) of the failed test and its related scenario file, and rerun it on the software simulator—thus greatly reducing debug efforts. Because FPGA verification doesn’t begin until the design is mature enough to run for a long time without too many quick failures, the resimulation cost when a test fails is small.

Coverage-driven methodology

Modern microprocessor verification tends to use a coverage-driven methodology to measure process closure. However, it’s difficult to measure coverage data in an FPGA environment. Therefore, in our VPOS-based method, coverage data is collected by profiling the code coverage of the reference ISS to reflect the DUT’s functional coverage. This is possible because a sequence of high-level descriptions in the behavioral ISS maps directly to a function in the DUT architectural specification.

When a test goes through the ISS for signature computation in the test generator, its coverage is obtained and fed back to the verification engineer, who modifies the constraints to direct the newly generated test and CCB to a specific (usually uncovered) functional point. Because the CCB is independent of the main test and is highly configurable, engineers can easily control what to verify and what not to verify, thus increasing the controllability of verification. Also, because the test’s efficiency is studied before the test runs on the FPGA, our method allows the design of a complete, golden test suite.

Results

We compared the results of FPGA verification of SLFs using VPOS and software simulation. To assess the efficiency of our method and sign off on the design, upon finishing the verification we ran Linux 2.4 and some small applications, the results of which we also compared with the FPGA.

Table 1 shows the bugs found at each stage. Software simulation accounted for most of the simple

<table>
<thead>
<tr>
<th>Verification stage</th>
<th>Bugs found (%)</th>
<th>Bug examples and descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software simulation with short tests</td>
<td>94</td>
<td>Simple bugs caused mainly by one SLF:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The state register is wrongly set to indicate a store access in a load translation lookaside buffer (TLB) miss exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- When an asynchronous on-chip timer interrupt occurs, the return-address register keeps the current program counter (PC), causing that instruction to be mistakenly re-executed when it returns from the interrupt.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- An attempt to write to an undefined special-purpose register should be treated as a no-op rather than raising an illegal exception.</td>
</tr>
<tr>
<td>VPOS-based FPGA verification running massive, long, random, and self-checking tests</td>
<td>6</td>
<td>Complex bugs that usually combine more than one SLF:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- When executing two consecutive branch instructions that both cause instruction TLB misses, the second TLB miss exception mistakenly preserves the previous machine state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- When a cross-page store access consecutively causes one instruction TLB miss, two data TLB misses, and one data protection violation, the prefetch logic doesn’t behave correctly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- When switching from translation mode to real mode, the load request of an atomic load/store pair instruction on an untaken branch is correctly canceled, but the reserve bit is mistakenly set.</td>
</tr>
</tbody>
</table>
bugs, whereas bugs detected in FPGA verification usually included complex address translations and exceptions that are difficult to cover using software simulation. These included page protection violations, translation lookaside buffer (TLB) misses, cross-page accesses, and very often a combination of these. This finding makes sense because directed-test writers might set up page tables to avoid long tests in software simulation and consequently miss some boundary conditions. The verification was successful, because no more bugs were captured with Linux 2.4.

Table 2 compares the debug efforts of VPOS with those of Linux. Booting and debugging Linux on a software simulator is time-consuming. Meanwhile, industrial practices and our past experiences indicate that debugging Linux on an FPGA using EDA vendor tools such as Altera’s SignalTap is also a nontrivial task, and the required time depends greatly on the engineer’s experience. So, whatever the case, a GPOS such as Linux is undoubtedly not suitable for debug, especially when the VPOS approach is skipped, leaving 6% of the bugs undetected. But such a GPOS can still serve as a good sign-off to mark the end of functional verification. In contrast, when the VPOS-based method is used, the time to generate, download, and run all tests on the FPGA without failure is 12 hours, whereas the debug time for only one failed test on a software simulator is 1 to 3 hours. The seamless environment in VPOS lets verification engineers take advantage of the FPGA's at-speed feature to run tests and move the costly debug work to software simulation, thus providing the greatest flexibility and the most predictable debug time.

Figure 7 shows the coverage curve for the VPOS-based tests and a Linux 2.4 simulation that we ran using the same reference ISS. The applications developed by our engineers are equivalent to the tests we ran in VPOS. To our surprise, coverage using Linux was less than 60%. The untouched paths involved SLFs such as misaligned interrupts. This is because the Linux operating system can optimize tests for software performance, which is not desirable when verifying hardware features. Moreover, it is obviously impossible to write tests in the Linux environment to cover a specific functional point, which can explain the slow advance of the coverage as the test count increased. With the VPOS method, on the other hand, we can obtain higher coverage, because all hardware features are under the verification engineer's control.

**OUR VPOS-BASED METHOD** is a new approach for the verification of microprocessor SLFs on FPGAs. In the future, we hope to extend our work in several directions. First, an important issue we are investigating is how to separate architecture-dependent CCB information from general descriptions of software scenarios. VPOS will then be able to handle a vast range of processor families with little change to architecture-specific information. Second, we are trying to automate coverage-driven test generation, especially the generation of SLF scenarios. Finally, we are preparing to extend VPOS so that it supports cache coherence scenario buildup and verification.
References


Lingkan Gong is a member of the research staff in the Department of IC Design at the East China Institute of Computer Technology. His research interests include test program generation for microprocessors, and methodologies for both software simulation and FPGA verification. Gong has an MS in computer science and engineering from the East China Institute of Computer Technology.

Jingfen Lu is a member of the research staff in the Department of IC Design at the East China Institute of Computer Technology. Her research interests include embedded-software support for verification, high-level modeling, and software-hardware codesign. Lu has an MS in automatic control from Xi An Jiao Tong University in China.

Direct questions and comments about this article to Lingkan Gong, Dept. of IC Design, East China Institute of Computer Technology, No. 418, Gui Lin Rd., Shanghai, China, 200233; glk@ecict.com.cn.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

IEEE Design & Test of Computers