Testing FPGA Devices Using JBits

Prasanna Sundararajan, Scott McMillan and Steven A. Guccione
Xilinx Inc.
2100 Logic Drive
San Jose, CA 95124 (USA)
{Prasanna.Sundararajan, Scott.McMillan, Steven.Guccione}@xilinx.com

Abstract

Unlike other integrated circuits, SRAM-based FPGA devices present a unique problem for testing. Because of their high level of configurability, it is often difficult to isolate and test device resources. This high level of configurability of FPGA devices, while a drawback for traditional test methods, can also provide unique advantages when performing device test. If configurability is used to tightly control the device functionality, individual components can be isolated and tested. One problem with existing FPGA test approaches is that they typically rely on traditional FPGA design tools to produce their test configurations and then use ASIC-style vectors to detect defects. With the availablity of new design tools which support low-level control and Run-Time Reconfiguration (RTR), detailed, architecture-specific tests can be written in existing high-level languages. This paper discusses the use of Xilinx's JBits(tm) Toolkit, to provide in-system test using run-time reconfiguration for the Xilinx Virtex(tm) family of devices.

1 Introduction

Testing of digital integrated circuits is a generally difficult task [1]. The goal in testing is to exercise all portions of the circuit, checking to ensure that all logical functions and interconnections are operating properly. In order to fully test a digital circuit, some input stimulus must be presented and a result read and compared to some predicted value. The need to completely test digital integrated circuits becomes difficult primarily because of the size and complexity of modern hardware devices.

If integrated circuit test is difficult, Field Programmable Gate Arrays (FPGAs) represent a special and particularly difficult type of digital circuit. Because FPGA devices are highly configurable, it is often difficult to isolate and exercise all elements of the architecture. In addition, modern FPGA devices represent some of the largest, most complex integrated circuit devices available. Because of this, most classical test techniques fail when applied to FPGA devices. This had led to a sub-speciality in the field of digital circuit test dealing specifically with FPGA devices.

While the configurability of FPGA devices leads to difficulties in testing using traditional methods, it is also possible to use these unique features to simplify testing. This paper describes the use Xilinx's *JBits Software Development Kit (SDK)* to provide both low-level device control to enable testing, as well as run-time reconfiguration to aid in both producing and evaluating test data.

2 Current FPGA Device Test Techniques

Testing of FPGA devices is currently very similar to the testing of any other integrated circuit device. In a production environment, standard, commercially available test machines are used to test FPGA devices. Devices are configured to produce circuits which are then tested using traditional test vectors. These test vectors are written to the device via external input pins and test results are typically extracted via other device output pins. This output data is then compared to expected results, either detecting errors or validating the correctness of the circuit. Figure 1 illustrates this device test approach.

Unlike fixed integrated circuits, FPGA devices have rich and variable architectural resources. No single configuration can exercise all of the capability of the device. So many re-configurations are performed, with new test vectors run for each device. The goal of the FPGA tester is to find the smallest number of such configurations which will exercise all elements of the device in the least amount of time. In a sense, because

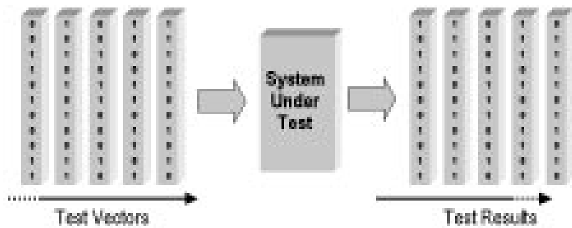

Figure 1: The standard device test approach.

the FPGA can behave like a wide variety of different devices, it must be tested while behaving like all of these different devices.

The problem is made somewhat simpler because the FPGA device is typically a repeated array of a small number of architectural elements. By breaking down the various elements into groups and testing them together, the problem becomes more manageable. For example, a circuit could be configured to test registers, Look-Up Tables (LUTs), distributed RAM, Block RAM (BRAM), or the Global Routing Matrix (GRM). The test vectors are designed to obtain as adequate coverage as possible for the resource element being tested. Recent work describing this approach by Renovell [2], [3], [4] gives a good description of this test methodology.

Another limitation of these test techniques is the amount of input and output bandwidth available to the device. This limits the ability to move test vectors and results on and off the device. As a result, the maximal coverage of defect detection decreases and the defect location will become more difficult to discern. Built-in Self Test (BIST) [5], [6] solves these issues by having the test vector generators and the output analysers inside the FPGA. However, this approach has additional overhead in internal FPGA memory required for storing test vectors and output results. In addition, BISTs force the location of the BIST to be defect free and require special software to access these BIST circuits.

Traditional test methods and BIST represent two ends of the test spectrum. Traditional test has the advantage that it uses standard tools and methodologies and requires no specific changes to the FPGA hardware. Unfortunately, this technique is restricted to the manufacturing floor and not well suited to testing devices in existing systems. Conversely, BIST techniques are more flexible and may even be used to test devices in-system. The drawback to BIST is that it requires special test circuitry to be designed into the FPGA device. These circuits can consume a substantial percentage of the device silicon, thus increasing the device cost.

3 Device Test Using JBits

Xilinx's *JBits Software Development Kit (SDK)* is a set of tools and Application Program Interfaces (APIs) for FPGA development. While the *JBits SDK* includes several high-level design tools, APIs and utilities to support activities such as run-time routing, debug and core-based design, these components are not of particular interest in addressing FPGA device test. Figure 2 gives a diagram of the components of the *JBits SDK* used in this work. First, and perhaps most important is the *JBits* configuration bitstream API [7]. This API permits device resources to be programmed and probed individually at run-time, even with the FPGA device in a working system. The other major component is the *XHWIF*(tm) portable hardware interface [8]. This provides a standard interface to Virtex-based hardware, permitting *JBits* applicaitons to be run across a variety of platforms, usually without re-compilation.

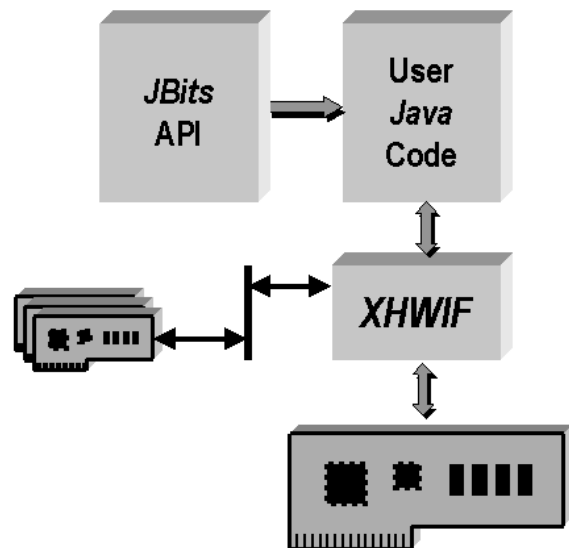

Figure 2: The JBits interface.

Apart from *JBits* and *XHWIF*, *JRTR* API [9] is also used in device testing. *JRTR*, using cache based approach, provides an effective model and implemen-

tion to support partial run-time reconfiguration. Partial run-time reconfiguration eliminates the overhead of configuring and reading back the full test configuration bitstream data, eventaully reduces overall testing time.

Because it can be assumed that CLBs are independent of each other with regard to defects, full coverage can be obtained by fully testing each CLB independently. Because of the run-time reconfiguration support in *JBits*, individual resourece in the device can be set and probed. This makes it possible to write a group of relatively small and simple applications that can test the entire device.

Unlike traditional approaches to FPGA device test, *JBits* makes active use of the reprogrammable features of the device to perform testing. Where current test strategies rely on static configuration bitstreams and external stimulation to perform testing, this approach permits dynamic modifications of internal circuitry to configure, drive and probe the section of the device under test. This can greatly reduce the bandwidth requirements of testing, both in the amount of configuration data necessary to provide coverage, as well as the number of test inputs or vectors required to stimulate the device.

4 Configuration Memory Test

The first and perhaps most important test is to verify that the configuration memory is working correctly. Currently, the largest FPGA devices have on the order of one million bytes of configuration data. Clearly, if all of the configuration data memories are not operating correctly, it will be dificult or impossible to proceed with testing. Aditionally, if these bits do not operate correctly, it is not only possible for a configured circuit to fail, but it is also possible that an illegal configuration could result, damaging the device itself or even other system components attached to the device.

While the configuration memory test is the first and perhaps the most important FPGA test, it is also perhaps the simplest using the *JBits* interface. Figure 3 shows the complete code sequence necessary to download a configuration bitstream to a device, read the configuration bitstream data back to a host processor, then perform a bitwise compare on the two configuration bitstreams.

In approximately six lines of code, a complete test of the configuration memory is performed. Note that for illustration the FPGA device type is hard-coded

```
/* Download configuration memory */
device = Devices.XCV800;
board.setConfiguration(device, bs1);

/* Readback configuration memory */
bs2 = board.getConfiguration(device, bytes);

/* Compare the data */
for (i=0; i<bs1.length; i++)
  if (bs1[i] != bs2[i])
    System.out.println("Bitstream diff
       in byte " + i);
```

Figure 3: JBits code to test the configuration memory.

to be a Xilinx Virtex XCV800. In a production environment, this device type would be a command line parameter. This would permit all devices in the Virtex family [10] to use this test without any recompilation. The configuration bitstream data files, of course, would necesarily be different for each device in the family.

It should also be noted that this code only makes use of the *XHWIF* portable hardware interface supplied with *JBits*. No actual manipulation of the bitstream is performed in the configuration memory test. Finally, this test should be repeated with several configuration bitstreams to completely exercise the configuration memory. Ideally, each bit of configuration memory will at some point be set to both a logic zero and a logic one.

5 Look-Up Table Testing

Once the configuration memory has been verified, it is possible to move on and test other architectural components of the device. Figure 4 gives the *JBits* code to test the Look-Up Tables (LUTs) in the device. Unlike the simple case of the configuration memory, this test makes use of features of the *JBits* toolkit, including the ability to dynamically create configuration data.

The sequence of operations in this code is again, relatively simple. Each of the F Look-Up Tables (LUTs) in Slice zero is set to zero. The configuration bitstream containing these zero-ed LUTs is downloaded to an FPGA device, in this case a Virtex XCV800, read back and checked. In less than 15 lines of code, LUT testing for bits stuck at one has been performed.

```
/* Set LUT values to zero */
jBits = new JBits(Device.XCV800);
clbRows=jBits.getClbRows();
clbCols=jBits.getClbColumns();
for (row=0; row<clbRows; row++)
  for (col=0; col<clbCols; col++)
    jBits.set(row, col, LUT.SLICE0_F, 0);

/* Download the configuration data */
bs1 = jBits.getAllPackets();
board.setConfiguration(device, bs1);

/* Readback the configuration data */
bs2 = board.getConfiguration(device, bytes);
jBits.setClbConfig(bs2);

/* Compare original and readback data */
for (row=0; row<clbRows; row++)
  for (col=0; col<clbCols; col++)
    if ((jBits.get(row, col, LUT.SLICE0_F) != 0)
      writeDefect(row, col, LUT.SLICE0_F);
```

Figure 4: JBits code to test LUTs.

Rather than run this test iteratively through all other groups of LUTs in the device, it is a simple matter to also test all of the other LUTs in the device for this defect in parallel by adding additional *JBits.set()* and *JBits.get()* calls and testing the result. Lastly, these LUTs can be tested for stuck-at zero defects by writing all ones to the LUTs and testing the read back results. As with the previous example, the device type and the LUT value can be passed as parameters, resulting in a more flexible test that will work for any device in the Virtex family for any LUT value test.

6 Testing Wires

While testing device components such as LUTs, flip-flops and other logic elements is fairly straight forward, testing wires is somewhat more difficult. Because modern FPGA devices contain such a large number of interconnection resources, this problem is not just one of complexity, but size as well.

One particular problem with testing wires is that existing FPGA design tools are not very good at explicitly controlling routing. This means that the configuration data generated for testing is either stochastic, using a variety of resources, or a wires must be painstaking isolated using low-level, often graphical, design entry tools.

While the approach used by *JBits* also relys on low-level access to routing resources, the API specifically allows this access in a relatively simple manner. In addition, because this low-level access is part of a layered API, access to higher level functions in *JBits* are still available.

Figure 5 gives a basic diagram of a configured circuit used to test a single wire in the device. Note that because other tests covering the configuration memory and logic components have already been performed, it is now possible to leverage these device components to test the remainder of the device.

In the figure, two four-input LUTs are configured as sixteen-bit shift registers (SRL16). The first SRL16 on the left is used to supply test data to drive the wire being tested. This SRL16 output passes through a switch, which in this case is simply a MUX and is used to drive the wire of interest. The data received on the other end of this wire are fed into the input of a second SRL16, again via a MUX for storage.

Typically, a single zero and a single one can be sent down the wire under test and shifted into the second SRL16. After the data has been used to test the wire, it can be read directly from the second SRL16 via the readback mechanism of the configuration port and compared to expected values. It the wire is broken or stuck at a value, this is detected.

Note that because of the two MUXes in the path of this test, it is possible that defects actually in the MUXes rather than in the physical wire may be detected by this type of test. For testing purposes, the MUXes are considered part of this wire. The "wire" in this case is taken to be part of the point-to-point connection between the first SRL16 output and the second SRL16 input. If a more detailed analysis is required, additional test may be run to further isolate the physical defect.

7 Performing Timing Tests

So far the tests described are used to isolate logic defects. The ability to test other device parameters is also possible using these techniques. Figure 6 shows a structure similar to that used for the testing of wires. In this case, however, device timing is characterized.

The circuit in Figure 6 uses an SRL16 sixteen bit shift register to send a test pattern of alternating ones and zeroes to a configured circuit, much as in the wire test circuit. Similarly another SRL16 is used to collect and store the results of the test. The configured circuit

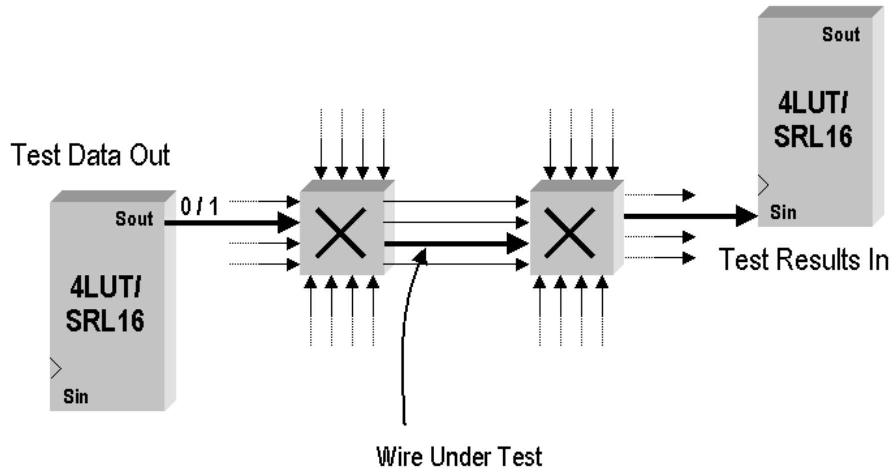

Figure 5: Testing wires using JBits.

under test in this case is a chain of LUTs configured as buffers.

Unlike the previous tests, this test requires a controllable clock with variable frequency. Data is sent from the source SRL16, across the configured chain of buffers and into the destination SRL16. This process is repeated several times per frequency, with the number of bits successfully transmitted recorded.

The graph in Figure 7 gives the result of this timing test for the F LUTs in both Slice 0 and Slice 1. The test was conducted on hardware containing a Virtex XCV1000 -4 speed grade device. 64 chains of 94 buffers were configured across the device. The clock speed was incremented in steps until the circuit failed completely. The test at each frequency was run ten times and the results averaged. For the F-LUT in Slice 1, the circuit begins to fail at approximately 6.25 MHz and has failed compeltely by 6.5 MHz. Because there are 94 buffers in the chains, this indicates a delay of approximately 1.72 nanosecond for each buffer. This delay does include not only the delay through the F-LUT, but also the delay through the single length wire used to connect the F-LUTs.

The graph also indicates that the Slice 0 F-LUT is somewhat faster than Slice 1 F-LUT, with a maximum frequency of 6.5 MHz achieved before failures are observed. This information was verified with the standard Xilinx tools, which also indicate a roughly five percent difference in speeds of the F-LUTs in Slice 0 and Slice 1. Also observed was slight but measurable variations of LUT speeds across the device. Some regions were consistently faster than others. This effect is still under investigation.

This example of testing the speed of LUTs within the device can easily be extended to testing the timing of other components, including flip-flops, wires and input / output resources. While these tests were conducted in-system in a relatively uncontrolled environment with respect to temperature, power supply voltage and other environmental factors, the techniques could also be used in a controlled enviroment for device characterization and grading.

8 Parallelizing the Tests

Because of the repetitive structure of FPGA devices, each CLB component can be tested across the device in parallel. The code in the LUT test in Figure 4 demonstrates that the addition of two lines of code, looping across rows and columns, parallelizes the testing task. In this case, the selected component in each CLB is tested in parallel. Note that these loops are only used to produce the configuration data. The looping overhead is not incurred as part of the testing; in fact, the configuration bitstream data may be written to a file and used off-line.

In addition, wires may also be tested in parallel in this fashion. In general, the number of tests required at this point is equal to the number of individual resources tested. However, further improvements are possible. Independent CLB components can also be tested in parallel. All LUTs, for instance can be tested in a single pass. Similarly, components such

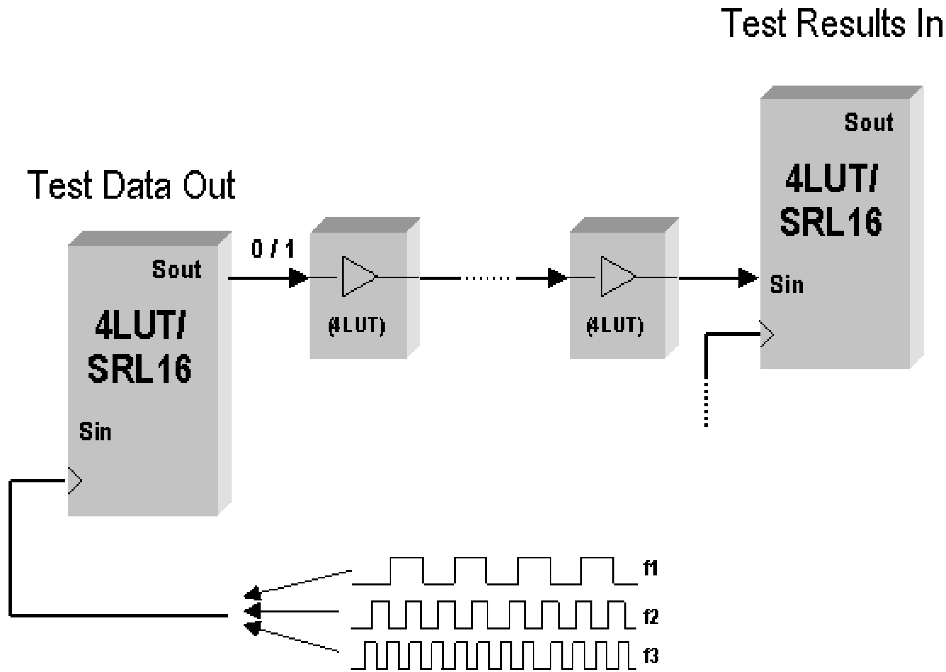

Figure 6: Performing timing tests using JBits.

as filp-flop set and reset can also be tested in parallel with LUTs.

In a similar fashion, wires may also be tested in parallel. A single SRL16 can be used to drive several wires, with results being stored in SRL16s in other CLBs at the end of the various wires under test. A single line to the north, south, east and west, for instance, can be tested in parallel this manner.

Finally, if all that is desired is a pass / fail test for a device, wires can be chained in regular patterns and tested in a single pass. If the patterns are localized, they may be repeated across the device and provide a rapid test for routing integrity.

9 Future Work

While this approach to testing complements the existing test techniques used in a production environment, it has several other uses. First, because it operates strictly through the device configuration port and requires relatively small amounts of code, this technique is ideal for in-system test. This can be used as part of a Power On Self Test (POST) sequence for systems which require this level of robustness. Additionally, it can be used as an on-line diagnostic tool for fielded systems. Coupled with very simple test equiment, *JBits*-based device testing can be used to help detect defects in-system.

In addition, the defect isolation capabilities of this technique can be used to produce a database of defects for a particular device. This database can be used to drive defect tolerant design tools such as those reported by Sundararajan and Guccione [11]. This permits otherwise rejected devices which fail testing to be used in functioning systems. And because this test technique is small and operates in-system, this approach can be taken on-line and provide true fault tolerance, resulting in self-repairable systems. Work in this area is in progress.

Work on device test is ongoing. First, work is proceeding to complete the suite of *JBits*-based device test routines, including tests for Block RAM and IOBs. In addition, other different types of tests are being explored. Specifically, tests to more completely characterize devices is being explored, along with tests to explore areas such as signal crosstalk and pattern sensitivity.

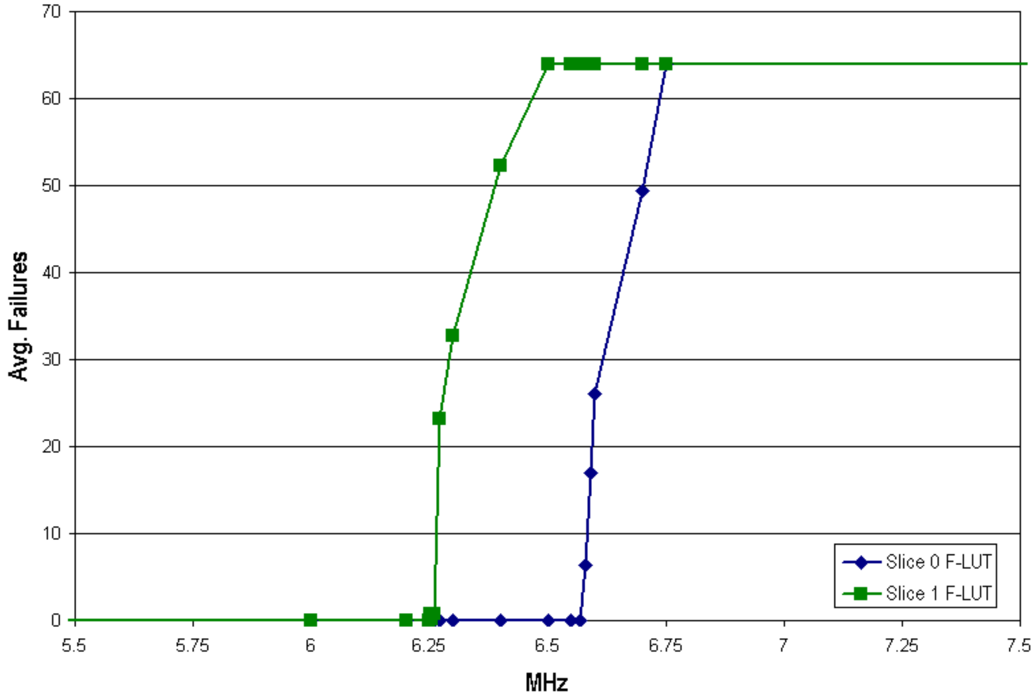

Figure 7: F LUT timings for Slice 0 and Slice 1.

10 Conclusions

The steps taken in this paper provide a proof of concept that *JBits* can be advantageous in testing FPGA devices. The ability to probe and stimulate internal states eliminates traditional testing concerns about I/O pad limitation and coverage requirements. This method also implements fast parallel testing with full coverage of all of the independent CLBs.

JBits removes the need for storing numerous bitstream files and test vectors. Small, simple code sequences are used to supply various tests in a device independent manner. In addition, *XHWIF* provides a standard interface for testing devices across different hardware platforms. These capabilities combine to make both manufacturing test floor and field testing viable.

Finally, *JBits* can be used to build accurate defect databases for individual devices in real-time. This is a necessary component for building fault tolerant FPGA circuits. These circuits can either be customized around known defects detected as part of the manufacturing process, or adaptively in fielded systems.

Acknowledgements

Thanks to Pavanish Nirula and other members of the Xilinx test group for helpful discussions on the practical aspects of FPGA device testing. This work was supported by the U.S. Defense Advanced Research Projects Agency, under contract DABT63-99-3-0004.

References

[1] Miron Abramovici, Melvin A. Bruer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* IEEE Press, Piscataway, NJ, 1990.

[2] M. Renovell. A specific test methodology for symmetric SRAM-based FPGAs. In Reiner W. Hartenstein and Herbert Gruenbacher, editors, *Field-Programmable Logic and Applications*, pages 300–311. Springer-Verlag, Berlin, August 2000. Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, FPL 2000. Lecture Notes in Computer Science 1896.

[3] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. RAM-based FPGAs: A test approach for the configurable logic. In *Proceedings of the 1998 Design Automation and Test in Europe (DATE '98)*, pages 82–88, Los Alamitos, CA, February 1998. IEEE Computer Society Press.

[4] M. Renovell, J. M. Portal, J. Figueras, and Y. Zorian. Testing the interconnect of RAM-based FPGAs. *IEEE Design and Test of Computers*, 15(1):45–50, January–March 1998.

[5] Charles Stroud, Eric Lee, and Miron Abramovici. Bist-based diagnostics of FPGA logic blocks. In *Proceedings of IEEE International Test Conference*, pages 539–547, 1997.

[6] Charles Stroud, Sajitha Wijesuriya, Carter Hamilton, and Miron Abramovici. Built-in self-test of FPGA interconnect. In *Proceedings of IEEE International Test Conference*, pages 404–411, 1998.

[7] Steven A. Guccione and Delon Levi. XBI: A java-based interface to FPGA hardware. In John Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 97–102, Bellingham, WA, November 1998. SPIE – The International Society for Optical Engineering.

[8] Prasanna Sundararajan, Steven A. Guccione, and Delon Levi. XHWIF: A portable hardware interface for reconfigurable computing. In John Schewel, editor, *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, Proc. SPIE 4525 (to appear)*, Bellingham, WA, August 2001. SPIE – The International Society for Optical Engineering.

[9] Scott McMillan and Steven A. Guccione. Partial run-time reconfiguration using JRTR. In Reiner W. Hartenstein and Herbert Gruenbacher, editors, *Field-Programmable Logic and Applications*, pages 352–360. Springer-Verlag, Berlin, August 2000. Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, FPL 2000. Lecture Notes in Computer Science 1896.

[10] Xilinx, Inc. *Xilinx Data Book*, 2000.

[11] Prasanna Sundararajan and Steven A. Guccione. Run-time defect tolerance using JBits. In *ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays (FPGA 2001)*, pages 193–198, February 2001.