# Processor Verification using Open Source Tools and the GCC Regression Test Suite: A Case Study

Jeremy Bennett, Embecosm

*This paper was presented at the Design Verification Club Meeting at Infineon, Bristol on 20 September 2010.*

**Abstract**

We present the OpenRISC 1200, an implementation of the OpenRISC 1000 architecture, verified using open source tools. We discuss a recent OVM verification project using SystemVerilog with SystemC.

A key part of processor verification is ensuring that the instruction set will correctly handle all the code it may ever be asked to run. The GNU Compiler Collection (GCC) tools include 53,000 tests of the C compiler alone. These provide an excellent test set of a processor's instruction set.

We report a case study using the OpenRISC 1200, in which this test set was used to verify the design in SystemC against its reference architectural simulation. The SystemC design model was built automatically from the implementation RTL using Verilator.

We conclude with results from this testing (which is still in progress) and make recommendations for the future OpenRISC 2000 project.

# 1 Introduction

The OpenRISC 1000 architecture defines a family of 32 and 64-bit RISC processors with a Harvard architecture [12]. The instruction set architecture (ISA) is similar to that of MIPS or DLX, offering 32 general purpose registers. The processor offers WishBone bus interfaces for instruction and memory access with IEEE 1149.1 JTAG as a debugging interface. Memory management units (MMU) and caches may optionally be included.

The core instruction set features the common arithmetic/logic and control flow instructions. Optional additional instructions allow for hardware multiply/divide, additional logical instructions, floating point and vector operations. The ALU is a 4/5 stage pipeline, similar to that in early MIPS designs.

The design is completely open source, licensed under the *GNU Lesser General Public License* (LGPL), this means it can be included as an IP block in larger designs, without requiring that the rest of the design be open source.

The OpenRISC 1200 (OR1200) was the first design to follow the OpenRISC 1000 architecture. It is a 32-bit implementation incorporating optional MMUs and caches, a tick timer, *programmable interrupt controller* (PIC) and power management. The implementation is approximately 32,000 lines of Verilog.

The overall design of the OR1200 is shown in Figure 1.

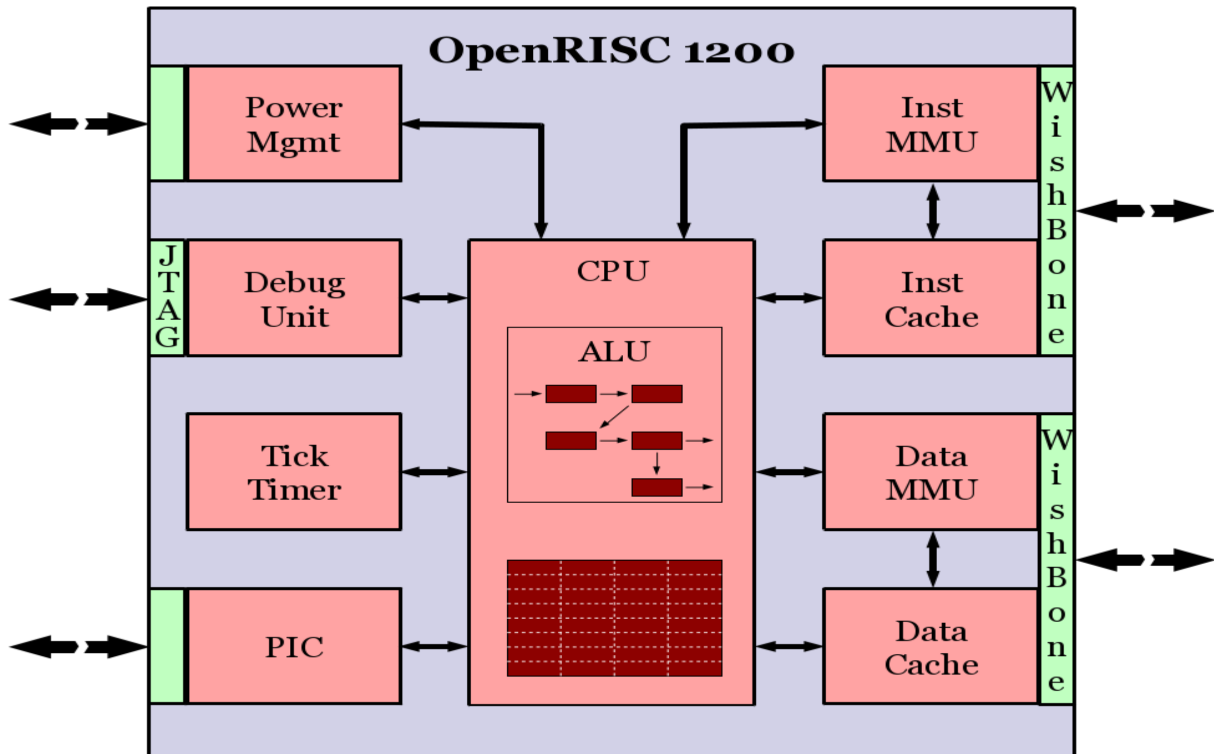


*Figure 1: Overall design of the OpenRISC 1200.*

# 2 Reference Golden Model

Or1ksim, the OpenRISC architectural simulator, is the golden reference for the OpenRISC 1000 architecture. Originally written in C, it now also offers a SystemC TLM 2.0 interface. It is a traditional interpreting *instruction set simulator* (ISS), which typically executes at 2-5 MIPS on a modern workstation.

Or1ksim also includes models of common peripherals components (UART, keyboard, VGA, Ethernet etc). It can be used standalone, as a GNU Debugger target (of which more later) or as a library within other programs.

Or1ksim is licensed under the *GNU General Public License* (GPL). This means that if it is incorporated within another program, that whole program will also be subject to the GPL.

## 2.1 Verifying the Golden Reference Model

Verification makes use of DejaGnu [5, 8], a POSIX 1003.3 compliant test framework. The tests are either native C programs which link to the library form of Or1ksim (to test the library interface) or C/assembler programs compiled with the OR1200 tool chain to exercise the functionality of the ISS.

A total of 2,275 tests of the ISS functionality and 264 tests of the library interface are included in Or1ksim at the time of writing (release 0.5.0rc1). These test the functionality of both the core ISS and peripherals. For floating point verification we make use of John Hauser's *TestFloat* software [7].

Or1ksim is built using the open source GNU autotools, *autoconf*, *automake* and *libtool* [9, 10, 11]. The tests can therefore be run using

```
make check
```

The results are reported as follows:

```
...
Running ../../testsuite/libsim.tests/int-edge.exp ...
Running ../../testsuite/libsim.tests/int-level.exp ...
...
Running ../../testsuite/libsim.tests/upcalls.exp ...

                === libsim Summary ===

# of expected passes            264
...
Running ../../testsuite/or1ksim.tests/basic.exp ...
Running ../../testsuite/or1ksim.tests/cache.exp ...
...
Running ../../testsuite/or1ksim.tests/tick.exp ...

                === or1ksim Summary ===

# of expected passes            2275
```

## 2.2 Limitations to the Golden Reference Model Verification

Although the Or1ksim test suite has expanded considerably in the past few months (it was originally just 20 tests), no effort has yet been put into ensuring all target functionality is exercised.

Furthermore no coverage metrics have been defined, nor coverage measured, so no assessment can be made of the quality of the coverage.

# 3 Verification of the OR1200 RTL using Icarus Verilog

The original verification OR1200 used a Verilog test bench which ran a number of test programs in C and assembler, compiled using the OpenRISC 1000 GNU C compiler.

Icarus Verilog is an open source event driven simulator after the fashion of Verilog XL. It runs between 20 and 50 times slower than modern commercial simulators such as Synopsys VCS, Cadence NC or Mentor Graphics ModelSim. It is capable of running the OpenRISC 1200 RTL at 1.4kHz on a 2GHz Core2 Duo E2180. The test bench comprises a total of 13 target programs.

The limitations of this approach are clear:

- it is not exhaustive;
- there are no coverage metrics; and
- the testing is not consistent with that of Or1ksim.

A medium term objective of the OR1200 design team is to unify this test bench with that of Or1ksim.

### 3.1 OVM Verification of the OR1200

For his MSc dissertation [1], Waqas Ahmed implemented an OVM testing regime for the OR1200. Using Or1ksim as golden reference he aimed to verify against 5 criteria, generating appropriate coverage metrics.

1. Does the PC update correctly?
2. Does the status register update correctly?
3. Do exceptions save context correctly?
4. Is data stored to the correct memory address?
5. Are results stored correctly in registers?

Although Ahmed used a commercial simulator for his work, he was able to make use of the SystemC interface to Or1ksim to implement a DPI SystemVerilog wrapper. The resulting test bench allowed comparative testing of Or1ksim against the RTL as show in Figure 2.

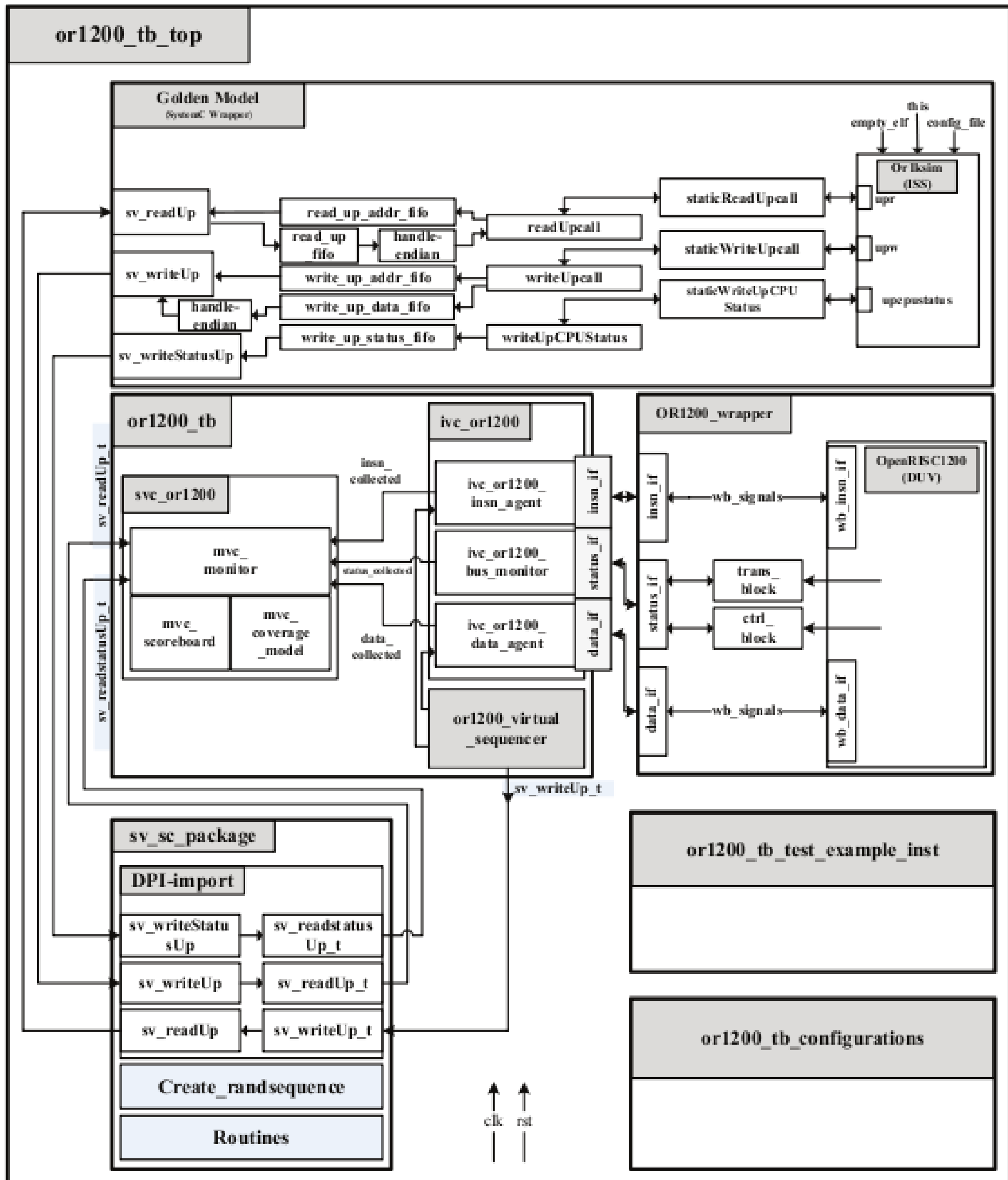


*Figure 2: Dual Or1ksim and RTL test bench.*

1. Discrepancies due to ambiguities in the architectural definition. An example being the handling of unaligned addresses by `l.jr` and `l.jalr`.
2. Instructions incorrectly implemented or missing in the RTL. Examples being `l.addic` and `l.lws`.
3. Instructions incorrectly implemented or missing in Or1ksim. Examples being `l.ror`, `l.rori` and `l.macrc`, although these are all optional instructions. However they are implemented in the OR1200 RTL.

In total 20 instructions had errors of some sort. This made for limitations in the coverage that could be achieved, since Ahmed did not have the opportunity to fix the RTL during the period of this project. However he was able to show that for many instruction set coverage criteria, he had achieved as full coverage as possible, while for others he had achieved significant coverage. There remain however a set of coverage criteria with 0% result, since all instructions in these cases had errors.

As a result of this work, the architectural specification has been updated, Or1ksim has been fixed, and changes to the RTL are in progress.

# 4 Verifying the OR1200 using GCC Regression Tests

The *GNU Debugger* (GDB) is a source level debugger for C, C++ and other languages. It may be used standalone or within Eclipse and can run native (i.e. debugging code on the machine on which it is executing) or with a separate embedded target. Most importantly for our purposes, its interface to a target is also used when testing all the components of the GCC tool chain.

GDB implements a simple packet protocol, the *Remote Serial Protocol* (RSP) to connect to an embedded target. Typically that will use TCP/IP to communicate to another process, which acts as a RSP server, and maps the packets into commands to drive the target, for example to JTAG via a FTDI 2232C USB interface.

Figure 3 illustrates this mode of operation.

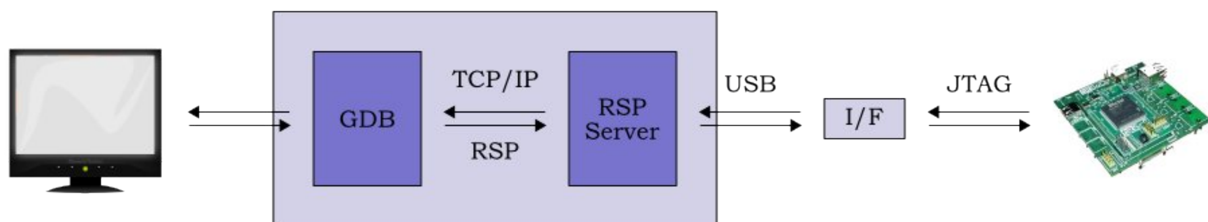


*Figure 3: GDB connecting to an embedded device via USB and JTAG.*

Once the RSP server is up and running, the user can connect from GDB using the *target remote* command.

```
(gdb) target remote 51000
```

## 4.1 Connecting GDB to a SystemC Model

A remote serial protocol server can just as easily be used to connect to a SystemC model. Rather than driving a physical interface through a library API, the RSP server drives the SystemC modeled JTAG interface. This may be a cycle accurate interface, or a transactional interface, as shown in Figure 4.

If the SystemC model is cycle accurate, then the interface between the RSP Server and SystemC model will use sc_signals to connect to **sc_in** and **sc_out** ports. If the model is transactional, then a TLM 2.0 transactional interface (loosely timed or approximately timed, blocking or non-blocking). The only slight issue is that JTAG is a bit serial interface, and some use of payload extensions is required to model this. For the GDB user, the interface is identical using the *target remote* command to connect, once the RSP server is up and running.

EMBECOSM™

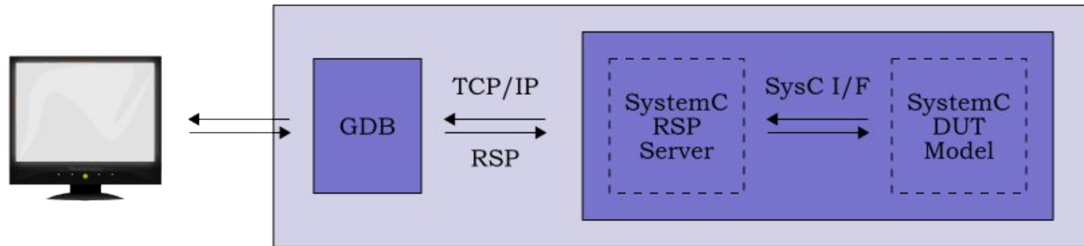


*Figure 4: GDB connecting to a SystemC model, incorporating RSP.*

As should be clear, the RSP server must itself be part of the SystemC model, and both it and the model of the embedded device form a single program.

## 4.2 Overall Class Structure of the SystemC Target RSP Server

The SystemC target RSP Server has four major components

1. *The RSP packet interface.* This handles the TCP/IP communications to send and receive packets to and from the GDB client. It requires no SystemC functionality, and has a procedural interface.
2. *The RSP analyzer.* This maps each packet into the appropriate set of actions (and responses) for the target processor's debug unit. It sends response packets back to the client as necessary. At its heart it is a large switch statement. A minimal implementation need only implement functions to read and write memory and registers, to set and clear breakpoints and to stall and unstall the processor. It is implemented as a SystemC module.
3. A model of the target processor's debug interface. This maps high level actions supported by the debug unit (read a register, stall the processor etc) into individual JTAG register transfers. It may be a separate SystemC module, or form part of the RSP analyzer
4. The target processor model. This may have a signal or transactional interface.

Figure 5 shows how these components make up the system.

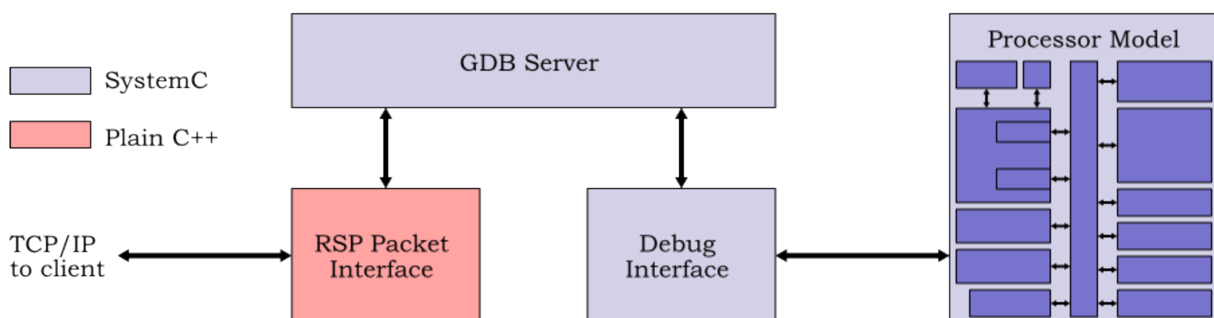


*Figure 5: Overview of a remote serial protocol interface to SystemC.*

By defining a transactional SystemC JTAG interface, we can separate out the implementation of the architectural interface to RSP (which is different for each architecture) from the implementation of the JTAG interface (which is different for each target interface type).

The standard TLM 2.0 generic payload cannot be used. It assumes that transactions are either read or write and that data is in byte sized chunks. JTAG is a simultaneous read and write and the data can be any number of bits. Furthermore there are only three possible addresses, the instruction register, the data register, and since we cannot cycle the state machine to a consistent position, a reset address.

The TLM 2.0 payload extension mechanism comes to the rescue here. We define an extension class, **JtagExtensionSC**, offering two additional attributes

- The type of JTAG transaction (**SHIFT_IR**, **SHIFT_DR** or **RESET**).
- The payload size in *bits*.

We can make this an ignorable extension, by treating addresses 0, 1 and anything else as respectively **SHIFT_IR**, **SHIFT_DR** or **RESET**, multiplying the byte size by 8 to get the bit size. The generic payload **tlm_command** is ignored—all JTAG transactions are combined write and read.

SystemC TLM 2.0 encourages the use of ignorable extensions for portability. However its value here is debatable. It is doubtful that any processor has a JTAG interface whose packets are always a multiple of 8 bits long.

The generic interface using this is illustrated in Figure 6.

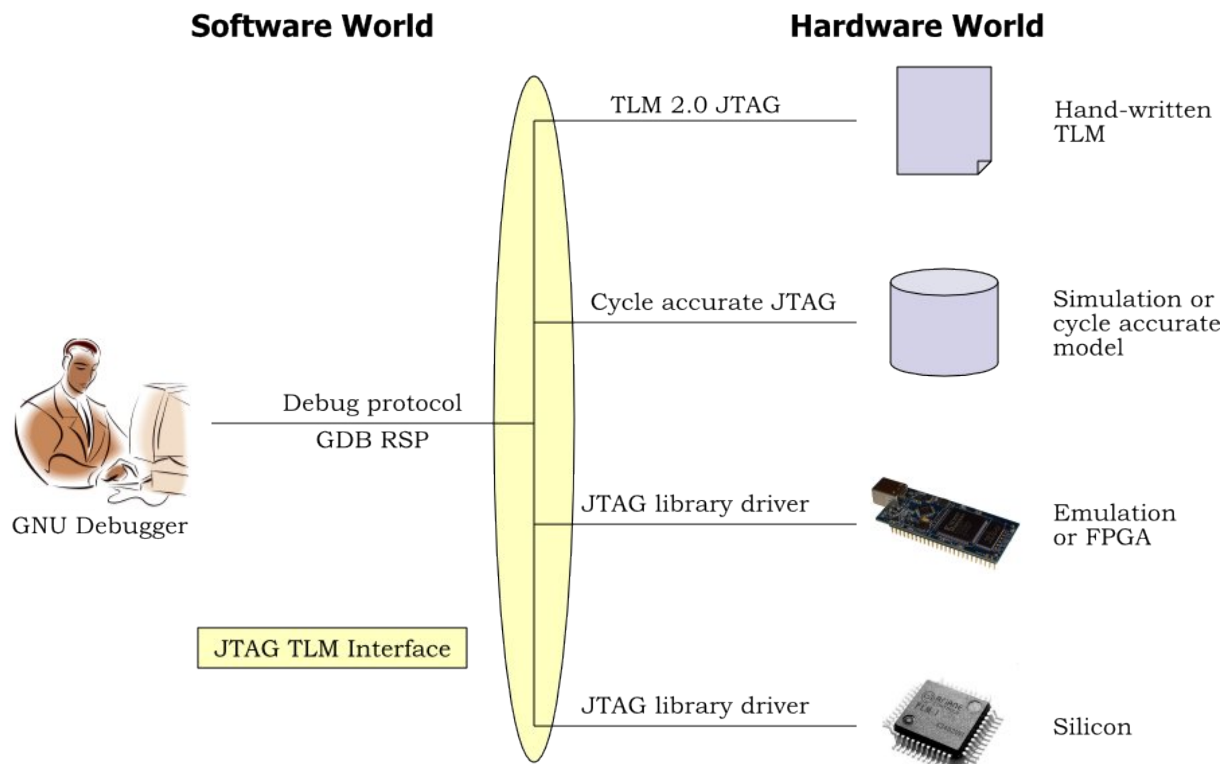


*Figure 6: Generic JTAG Debug Interface to SystemC.*

More detail on this approach was presented to the European SystemC Users Group at their meeting in September 2010 [6].

### 4.3 GCC Tool Chain Regression Testing for Verification

As noted, integrating a SystemC model with GDB allows the model to be exercised using the standard GCC regression test suite.

This provides a useful addition to standard regression testing. The tests can be run against a golden reference TLM and cycle-accurate model generated from RTL for comparison. The tests exercise the processor in a way that is relevant (running compiled code), with tests, which over a period of 20+ years have been designed to exercise processors where they are most likely to break.

These tests have some properties that make them particularly suited to our purpose.

- They test how well the processor runs C/C++ code, so exercise the architecture in ways that are particularly relevant to its use.
- They test corner cases that have been bugs in the past, that is areas which have caused problems with other architectures.
- They use the same interface as GDB to drive the target, so can be used with our interface to drive both the architectural model Or1ksim and the RTL as a Verilator model.

The tests run under the control of DejaGNU making the process easy to control.

These test sets are large. For the latest version of the major tools at the time of writing we have the following.

- GNU C compiler: approx 53,000 tests
- GNU C++ compiler: approx 20,000 tests
- GNU C++ library: approx 3,000 tests
- GNU Debugger: approx 10,000 tests

Not all these tests involve execution (there are tests of compiling, linking and so on), but they still represent a substantial load on the processor.

### 4.4 Regression testing the OpenRISC 1000

We first ran our regression tests against Or1ksim using just the core C tests for GCC 4.5.1. The results are as follows.

```
                === gcc Summary ===

# of expected passes            52753
# of unexpected failures        152
# of expected failures          77
# of unresolved testcases       122
# of unsupported tests          716
```

There are some failures, since this is a GCC compiler that is still not fully developed. As noted above, many of the tests are for compiling and linking. However there are a total of 7,916 tests which execute code and give a result. The total run time for the test suite was 6,936 seconds on a 2GHz Core2 Duo E2180.

For comparison we ran the same tests against the RTL, converting it to a cycle accurate SystemC model using Verilator. With full compiler optimization and

branch-prediction enabled, it is possible to achieve 130kHz on the same workstation. However for the purpose of these tests we used simple -O2 optimization, yielding a model capable of around 90kHz.

Rerunning the same test suite, the results are as follows.

```
                    === gcc Summary ===

    # of expected passes            52677
    # of unexpected failures        228
    # of expected failures          77
    # of unresolved testcases       122
    # of unsupported tests          716
```

The total run time was 19,099 seconds. It may seem surprising that the run time has less than tripled, given the model is running 20-50 times slower. However the majority of the tests are very short in execution time and only 15% of the tests are of execution anyway (the rest being compiling and linking). Because the Verilator model reports the number of cycles executed upon completion, we can determine the total number of cycles executed by the model, which is 4,977,311,609.

There are more failures. Some of these are tests which gave a different result, but 51 are tests which timed out. We are interested in those tests which passed with the golden reference and which failed with the RTL model. We can identify three causes of these new failures.

1. The test timed out, because the SystemC model based on RTL is 20-50 times slower than the TLM model, and this was a long running test which just needed more processor time.
2. The test timed out, because the model hit a problem and would never terminate.
3. The test returned a different result.

The second and third of these are indicators of possible bugs in the RTL. They provide test case code, which the design engineer can then use to isolate implementation errors.

The DejaGNUtest suite produces a comprehensive log of the tests which have failed. Thus we can identify that the test labeled:

```
gcc.c-torture/execute/20011008-3.c execution, -O0
```

timed out. A manual rerun of the command line in the log shows that this test does complete if given enough time—it just requires 115 million cycles. Inspection of the code shows it contains large nested *for* loops, so this is not surprising. This is an example of the first class of failure which does not indicate any problem with the RTL.

However the following test also times out.

```
gcc.c-torture/execute/20020402-3.c execution,  -Os
```

Manual rerunning does not allow this test to complete, even after two hours. In any case inspection of its sibling tests with different optimizations show they only require a few hundred thousand cycles to complete. This is an example of the second type of failure. At this point we have a clear test case for an RTL failure. Typically we will run the test with tracing enabled using both RTL and Or1ksim

versions, and determine where execution diverges. A VCD inspection usually quickly shows the cause of the failure.

Finally the following test completed execution, but gave the wrong result.

```
gcc.c-torture/execute/20090113-1.c execution,  -O2
```

In this case the test terminated with a Bus Error exception. This is an example of the third type of failure.

There is a fourth type of failure possible, which is where a test passes in RTL, but fails with Or1ksim. No such failures have been found to date, but they could indicate Or1ksim incorrectly implementing the architectural specification.

### 4.5 Further Testing

This paper represents work in progress. At the time of writing we are still working through the discrepancies to identify the causes of the errors and thus fix the RTL. When this is complete, we will repeat the exercise with the C++ tests, C++ library tests and GDB tests.

### 4.6 Commercial Adoption

Integrating SystemC models with the GNU debugger has been a standard part of the OpenRISC development environment for some time [12].

More recently, Adapteva Inc worked with Embecosm to integrate GDB with Verilator models of their new processor pre-silicon. These models were subsequently used to test the Verilog using the GNU tool chain regression suite as described above, and identified 50-60 RTL errors, which were able to be fixed before tape out.

## 5 Conclusions

The original OR1200 verification was inadequate. Running a handful of test programs through a processor is not sufficient.

An OVM test approach was helped by open source tools. SystemC made for easy SystemVerilog integration of Or1ksim. This allowed identification of numerous bugs and also gave us our first coverage data.

Integrating models to GDB through SystemC is easy. SystemC TLM 2.0 abstraction makes for easy reuse, allowing the same interface to drive a transactional architectural model and a cycle accurate model derived from RTL.

The GNU tool chain regression test suite is a valuable tool for exercising a processor design with a large amount of representative code. Comparative regression identifies possible RTL errors. In the case of the OpenRISC 1200, we have identified 76 discrepancies, which we are currently analyzing in detail.

## 6 The Future

The OpenRISC 2000 project has just started. Is this an opportunity to get things right first time?

EMBECOSM™

# 7 Further Reading

Embecosm has published a number of application notes, which address the issues of modeling processors using TLM 2.0 [2], using JTAG with SystemC [3], integrating GDB with SystemC models [4] and using DejaGNU[5].

# 8 Acknowledgments

Julius Baxter of ORSoC AB worked with the author to integrate Verilator SystemC modeling into the standard OpenRISC 1000 design flow.

# 9 References

1 Waqas Ahmed. *Implementation and Verification of a CPU Subsystem for Multimode RF Transceivers*. MSc dissertation, Royal Institute of Technology (KTH). May 2010.

2 Jeremy Bennett. *Building a Loosely Timed SoC Model with OSCI TLM 2.0: A Case Study Using an Open Source ISS and Linux 2.6 Kernel.* Embecosm Application Note 1, Issue 2, May 2010.

3 Jeremy Bennett. *Using JTAG with SystemC: Implementation of a Cycle Accurate Interface.* Embecosm Application Note 5, Issue 1, January 2009.

4 Jeremy Bennett. *Integrating the GNU Debugger with Cycle Accurate Models: A Case Study using a Verilator SystemC Model of the OpenRISC 1000.* Embecosm Application Note 7, Issue 1, March 2009.

5 Jeremy Bennett. *Howto: Using DejaGnu for Testing: A Simple Introduction*. Embecosm Application Note 8, Issue 1, April 2010.

6 Jeremy Bennett. Using SystemC Processor Models with the GNU Debugger. 22nd meeting of the European SystemC Users Group, Southampton 14 September 2010.

7 John Hauser. *TestFloat*. www.jhauser.us/arithmetic/TestFloat.html.

8 Rob Savoye. *DejaGnu: The GNU Testing Framework*. Free Software Foundation, 2004. Available at www.gnu.org/software/dejagnu/manual.

9 *The autoconf project*. www.gnu.org/software/autoconf.

10 *The automake project*. www.gnu.org/software/make.

11 *The libtool project*. www.gnu.org/software/libtool.

12 *The OpenRISC 1000 project*. opencores.org/openrisc,overview.

### About the Author

**Dr Jeremy Bennett** is Chief Executive of Embecosm Limited (www.embecosm.com). We want our users to develop embedded software seamlessly, using standard open source tools, whether the target is an early model of the architecture or final silicon.

Embecosm's services include:

- Comprehensive GNU tool chain porting and optimization for embedded processors.

- Standards based cycle accurate and transaction level hardware modeling, including OSCI SystemC TLM 2.0 compliance.
- Seamless, unified SoC firmware development and debugging from initial model to final silicon.
- Support, consultancy, tutorials and training throughout the product life cycle

Jeremy Bennett is an active contributor to the OpenCores project (www.opencores.org). Contact him at jeremy.bennett@embecosm.com.

*This paper was presented at the Design Verification Club Meeting at Infineon, Bristol on 20 September 2010.*

## Licensing

This work is licensed under the Creative Commons Attribution 2.0 UK: England & Wales License. To view a copy of this license, visit creativecommons.org/licenses/by/2.0/uk/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

This license means you are free:

- to copy, distribute, display, and perform the work
- to make derivative works

under the following conditions:

- *Attribution.* You must give the original author, Jeremy Bennett, credit;
- For any reuse or distribution, you must make clear to others the license terms of this work;
- Any of these conditions can be waived if you get permission from the copyright holder, Embecosm; and
- Nothing in this license impairs or restricts the author's moral rights.