

Using SystemC Processor Models with the GNU Debugger

Jeremy Bennett, Embecosm

This paper was presented at the 22nd European SystemC Users Group Meeting at the School of Electronic and Computer Science, Southampton University on 14 September 2010.

Abstract

It is common to use SystemC models of processors today. These may be hand-written architectural models or cycle accurate models generated using tools such as Verilator from design RTL.

We show how a SystemC model can be integrated with the *GNU Debugger* via its simulation interface. The user is then able to load, execute and debug programs on the SystemC model using the full range of GDB commands.

A follow on from this is that the integration allows the entire GNU regression test suite to be run against the model, providing a comprehensive exercise of the SystemC model. This is an important addition to the verification of the ISA in new processor designs.

We illustrate the technique with examples from the OpenRISC 1000. Our work has identified a limitation of SystemC when used as a subsidiary library, and we propose that this be remedied in the future.

1 Introduction

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.

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 1 illustrates this mode of operation.

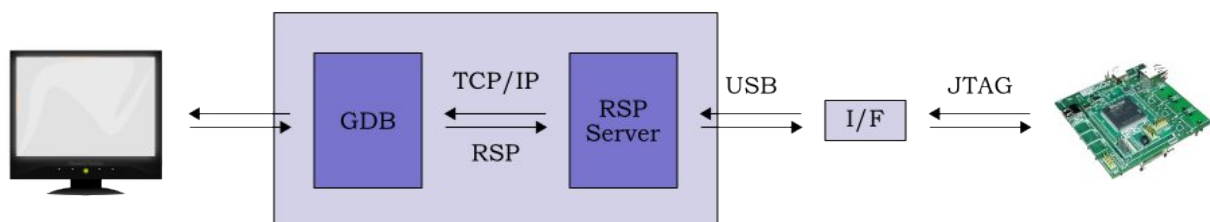


Figure 1: GDB connecting to an embedded device via USB and JTAG.

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

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

2 Using a SystemC Model as Remote GDB Target

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 2.

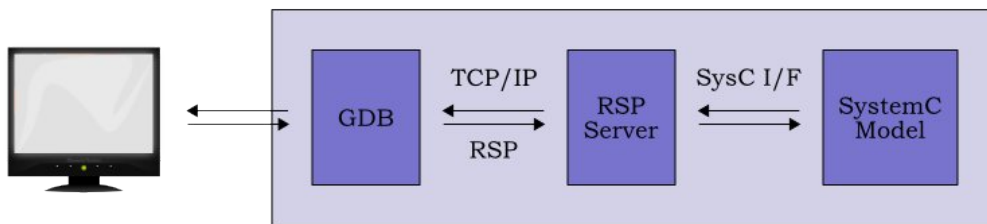


Figure 2: GDB connecting to a SystemC model of an embedded device.

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.

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, and is more accurately shown in

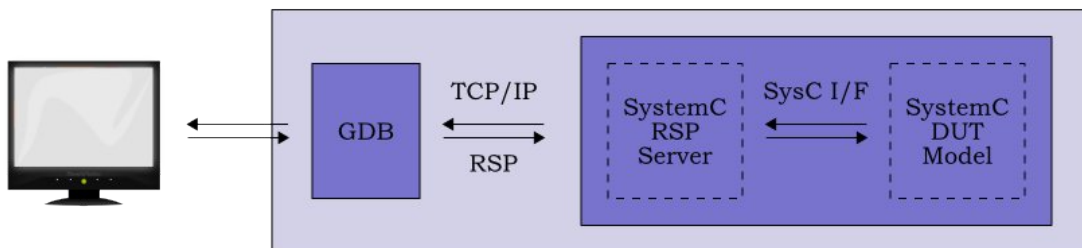


Figure 3: GDB connecting to a SystemC model, incorporating RSP.

2.1 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 4 shows how these components make up the system.

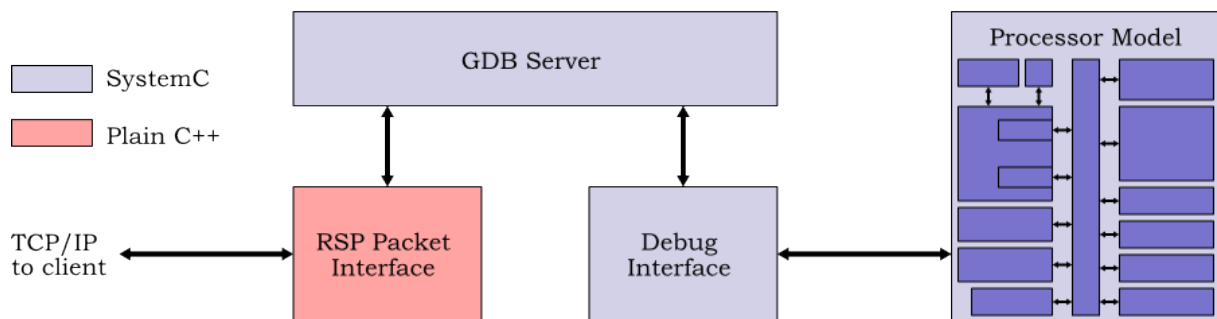


Figure 4: Overview of a remote serial protocol interface to SystemC.

2.2 Interfacing GDB to Cycle Accurate SystemC Targets

A common way to generate a SystemC model is from the Verilog RTL using a tool such as Verilator. We can interface the RSP server to its cycle accurate JTAG ports. The OpenRISC 1000 offers just such a model.

IEEE 1149.1 JTAG is a shift-register interface. For debug units, a register is shifted in to cause an action, with the result read from the same register as it is shifted out. The standard defines two register types (instruction and data) and the low level protocol for shifting registers.

Interpretation of the register is up to the target processor, although the standard does provide a standard interpretation for some instruction register sequences. Registers may be any number of bits long, and indeed the length of any individual data register may be determined by the initial bits shifted in.

The interface requires 4 wires, **TDI** (input, the bit being shifted in), **TDO** (input, the bit being shifted out), **TMS** (input, controlling the JTAG state machine) and **TCK** (the clock, an output, since the clock rate is determined by the target). These can be modeled in SystemC using **sc_in** and **sc_out** ports, connected via **sc_signal**.

Internally the processor model will shift in the bits, and when the sequence is complete, latch the full value for transfer to the debug unit. Figure 5 shows the state machine (defined by IEEE 1149.1) which controls this.

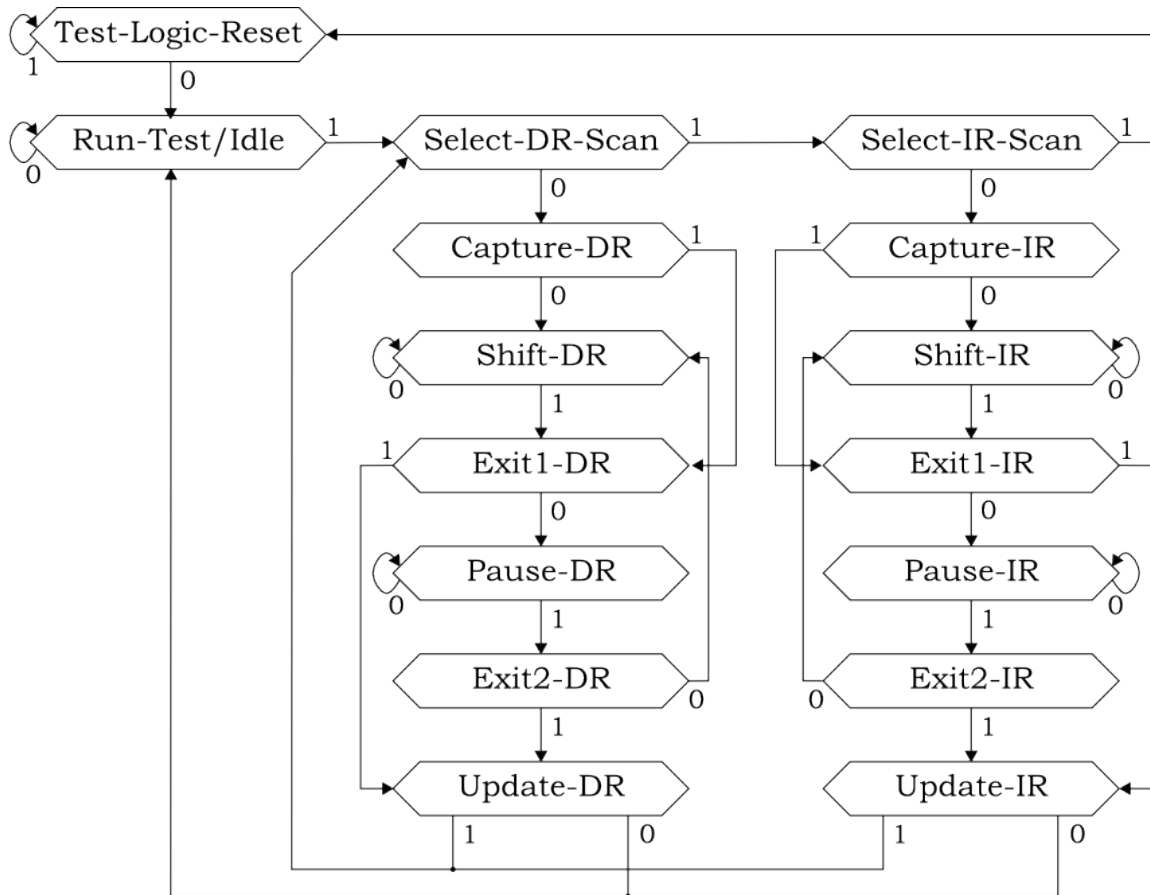


Figure 5: IEEE 1149.1 JTAG Test Access Port state machine.

The SystemC debugger must run a mirror of this state machine. Driven by the TCK signal from the processor, and the data from the debug interface, this will generate the TMS and TDI signals to send to the processor, and correctly accept the TDO signal being received.

Thus the debug interface is acting effectively as a *transactor*. Taking a high level operation (a “transaction”) and mapping it into a sequence of signals.

The system is complete. Starting the target model we wait for the GDB client to connect.

```
./Vorpsoc_fpga_top
```

```
SystemC 2.2.0 --- May 16 2008 10:30:46
Copyright (c) 1996-2006 by all Contributors
ALL RIGHTS RESERVED
```

```
Loading flash image from sim/src/flash.in
(orpsoc.v.uart_top) UART INFO: Data bus width is 32. Debug Interf
```

```
(orpsoc.v.uart_top) UART INFO: Doesn't have baudrate output
```

```
Listening for RSP on port 51000
```

We can now connect from the GDB client.

```
$ or32-uclinux-gdb
Building automata... done, num uncovered: 0/216.
```



```
Parsing operands data... done.
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses
This is free software: you are free to change and redistribute it
There is NO WARRANTY, to the extent permitted by law. Type "show
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=or3
(gdb) file hello
Reading symbols from ../sw/test-progs/hello...done.
(gdb) target remote :51000
Remote debugging using :51000
0x040001f0 in ?? ()
(gdb) load hello
Loading section .text, size 0x1350 lma 0x0
Loading section .rodata, size 0x1f lma 0x1350
Start address 0x100, load size 4975
Transfer rate: 323 bytes/sec, 236 bytes/write.
(gdb) continue
Continuing.
Remote connection closed
(gdb)
```

Meanwhile the target model has run to completion having hit an exit opcode.

```
Hello World!
The answer is 42
546960700.00 ns: Exiting (42)
SystemC: simulation stopped by user.
Closing connection
real 2708.49
user 87.82
sys 0.53
```

Note that the output is generated by the target model, and appears on its screen, *not* in the GDB session.

2.3 Interfacing GDB to Transactional SystemC Targets

Many processors are now available as high level SystemC models offering TLM 2.0 interfaces. The OpenRISC 1000 Or1ksim architectural simulator can be used in this way with a loosely-timed TLM 2.0 blocking interface.

The overall RSP server structure is unchanged, but this time, instead of driving a transactor to generate JTAG signals, the debug interface generates JTAG transactions.

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.

With this interface used by both the RSP debug interface, and the target we have a working GDB interface. The usage is identical to the cycle accurate version, but the user will notice a substantial increase in performance, due to the faster target model.

2.4 A Generic GDB to SystemC Interface.

In the two previous examples we were able to reuse a great deal of code. Indeed it was only the mapping from debug interface commands to JTAG that needed to be rewritten.

The definition of a transactional JTAG interface, allows us to make this a generic solution, and maximize reuse. This is illustrated in Figure 6.

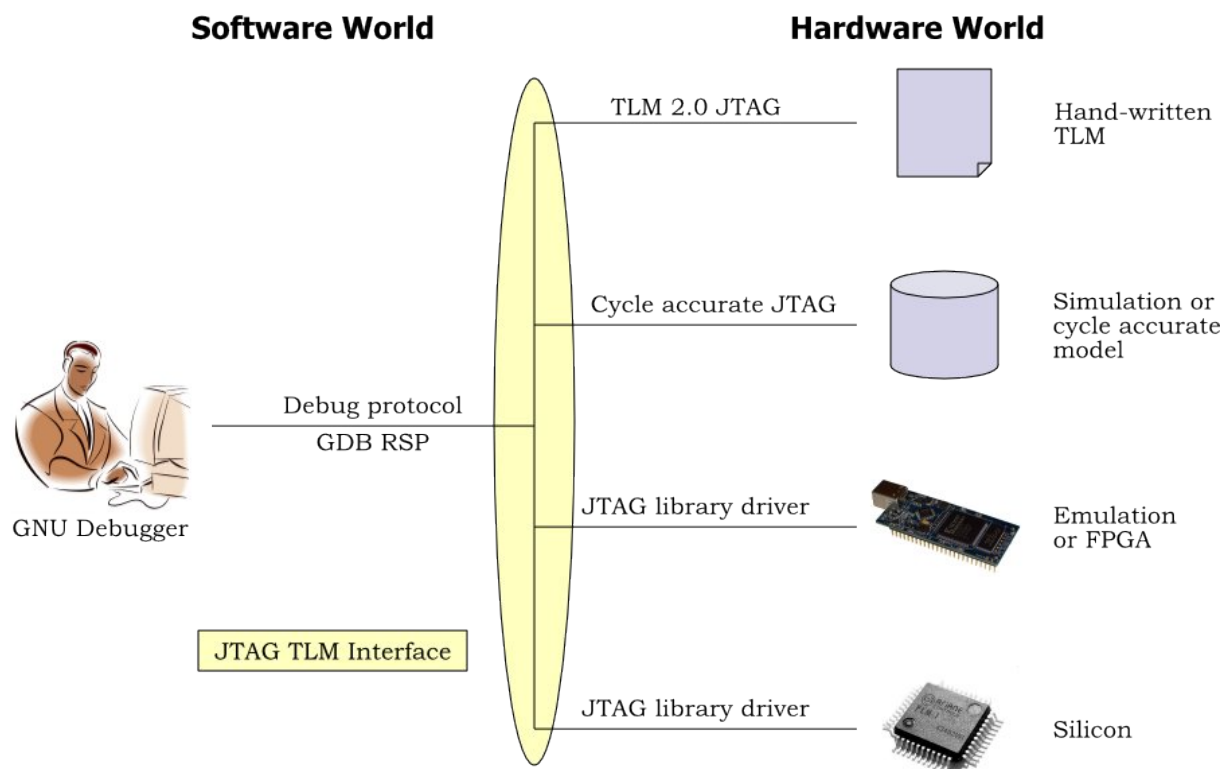


Figure 6: Generic JTAG Debug Interface to SystemC.

The top level class diagram is very simple. A RSP server class (**RspServerSC**), performing the architecture specific analysis, communicating using a generic payload with extensions to a transactional JTAG interface (**JtagSC**), which then drives the target. A class (**JtagRegister**) is defined to hold the information needed to populate the generic payload. This is shown in Figure 7.



Figure 7: Top level class diagram for a unified RSP to SystemC JTAG interface.

We can then add a set of transactors, which map the JTAG TLM interface to specific implementations as shown in Figure 8.

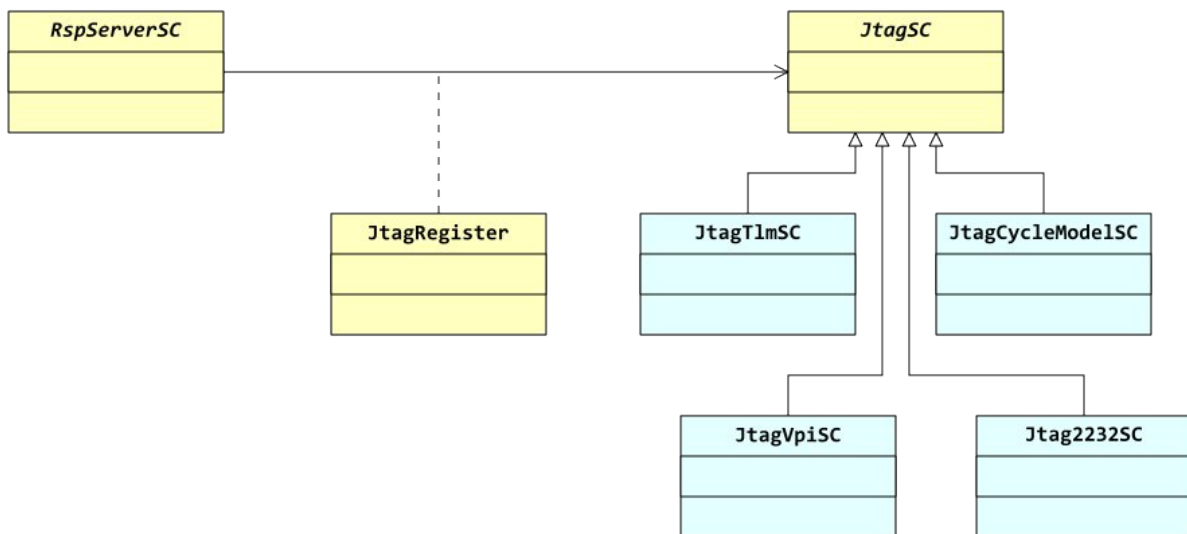


Figure 8: Derived classes for different SystemC JTAG interfaces.

This provides an ideal separation. The transactors only need to be written once for each type of interface (loosely timed TLM, approximately timed TLM, cycle accurate SystemC, VPI simulation interface, FTDI 2232C USB etc).

Similarly for each new architecture, the RSP analyser need only be written once. As a derived class it need only implement new architecture specific functionality. For convenience we can define derived classes to represent the different types of JTAG registers that are used (for example to access memory, access registers, control the CPU). This is shown in Figure 9.

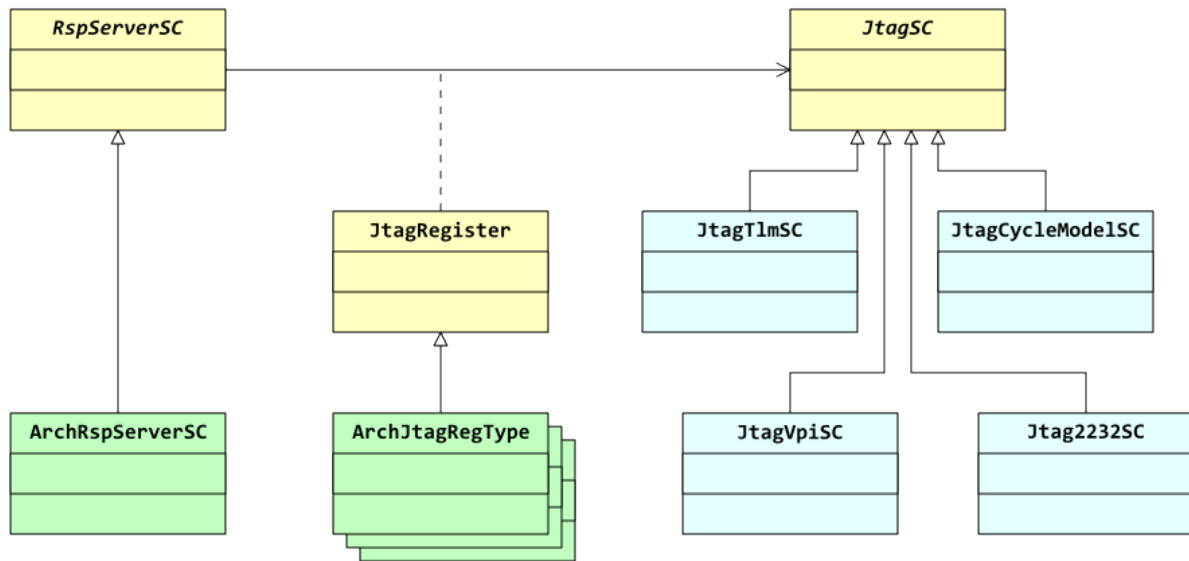


Figure 9: Derived remote serial protocol classes for different target architectures

The result is that having implemented the architecture specific functionality at an early stage, the user has a full RSP server implementation that can be used with no further work throughout the project, from initial model through to final silicon.

3 The Integrated GDB Simulator

There is a drawback to this approach to integrating GDB with SystemC models. The target is a separate process, with its output in a separate window. The user issues GDB commands in one window, and the output from the program (in this case “Hello World!”) appears in a different window. It would be possible to merge the outputs, but wouldn't it be better if the simulator were fully integrated into GDB.

GDB supports the concept of integrated simulators. It provides a command, **target sim**, and an interface that simulators should implement. The user can then use the simulator target from within GDB.

```

(gdb) file hello
(gdb) target sim
(gdb) load
Loading section .vectors, size 0x2000 vma 0x0
Loading section .init, size 0x2c vma 0x2000
Loading section .text, size 0xdf84 vma 0x202c
Loading section .fini, size 0x28 vma 0xffb0
Loading section .rodata, size 0x88c vma 0xffd8
Loading section .ctors, size 0x8 vma 0x12864
Loading section .dtors, size 0x8 vma 0x1286c
Loading section .data, size 0x888 vma 0x12874
Start address 0x202c
Transfer rate: 559072 bits in <1 sec.
(gdb) run
Starting program: /home/jeremy/svntrunk/GNU/or32/progs_or32/a.out
Hello World
  
```



```
Program exited normally.  
(gdb)
```

This approach provides an added benefit that the simulator is also compiled as a standalone program, **run**, allowing programs to be tested from the command line, thus.

```
$ or32-elf-run hello  
Hello World  
$
```

The beauty of this is that the **run** command is used by the entire GNU tool chain for its regression testing (although it is possible to use remote targets as well). So if we can integrate our SystemC model into the GDB simulator, we immediately have an easy way of testing our target model with the entire GNU tool chain regression suite.

3.1 Implementing the GDB Simulation Interface

The immediate challenge is that GDB is a large (1M+ lines¹) C program, while SystemC is a C++ library. However linking C and C++ is solvable, by giving the C++ program a wrapper layer with C linkage. It is also quite feasible to pass additional libraries to GDB for linking, although since the final linking step has to be with C++ and not C, some care is needed.

However the real problem is with the semantics of the interface. The key functions to be implemented are.

- **sim_open**. Instantiate the simulator.
- **sim_close**. Free the simulator.
- **sim_resume**. Single step or run the simulator until it hits a breakpoint or halts.

Other functions provide for access to registers, memory etc.

The problem is that we can call the function **sc_elab_and_sim** to initiate SystemC execution, but it will call **sc_main** and run to completion. What SystemC lacks is any mechanism to suspend execution and return control temporarily to the caller. In effect a SystemC model can never be a library component, it must always be the main program.

The only solution the author has found to date is to fork a process to run the SystemC model and communicate with it via a protocol such as RSP. The link and protocol can be simplified, but we still need two processes.

This is a practical solution, which has been implemented by the author and used both to run GDB sessions (using **target sim**) and to run GCC regression tests (using the **run** command).

A future version of SystemC would benefit from a mechanism which allowed control to be suspended.

¹ Source lines in GDB 7.2 **gdb** and **sim** directories as measured by John D Wheeler's *sloccount* program.

4 GNU Tool Chain Regression Testing for Verification

As noted, integrating a SystemC model with GDB, either as a standalone simulator, or as a separate program communicating through the remote serial protocol allows the model to be exercised using the standard GNU 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.

This is not the primary subject of this talk, and is discussed in more detail in a talk by the author to be given to the UK Design Verification Club later this month. However a brief example will show the value of this approach.

4.1 Regression testing the OpenRISC 1000

Having built our simulator into GDB, we can run our regression tests. The core C tests for GCC 4.5.1 against the reference OpenRISC 1000 TLM model (Or1ksim) 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. Not all the tests involve executing code—many are tests of compilation and linking. However there are a total of 7,916 tests which execute code and give a result.

We can then compare this against the same test suite run using a SystemC model generated from the OpenRISC 1000 Verilog RTL using Verilator.

```
=== gcc Summary ===
```

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

There are more failures. Some of these are tests which gave a different result, and 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.

5 Commercial Adoption

Integrating SystemC models with the GNU debugger has been a standard part of the OpenRISC development environment for some time [4].

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.

6 Conclusions

Integrating a SystemC model of a processor as a GDB remote target using the remote serial protocol is a straightforward task. SystemC provides the functionality required for both TLM and cycle accurate models.

The class inheritance mechanism of C++ when combined with SystemC's TLM 2.0 infrastructure allows a generic debug interface mechanism to be constructed. This minimises the effort in implementing a GDB interface to a model for a new architecture.

Integrating SystemC models within GDB directly is more of a challenge. To do this efficiently requires SystemC to be usable as a library interface. This in turn requires that SystemC can suspend and return control to a caller mid-simulation.

A consequence of integrating with GDB is that hardware design verification can take advantage of the GNU tool chain regression suite for hardware testing.

7 Further Reading

Embecosm has published a number of application notes, which address the issues of modeling processors using TLM 2.0 [1], using JTAG with SystemC [2] and integrating GDB with SystemC models [3].

8 Acknowledgements

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

9 References

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



4 *The OpenRISC 1000 project.* <http://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 22nd European SystemC Users Group Meeting at the School of Electronic and Computer Science, Southampton University on 14 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.