Who Ate My Battery?

Why software engineers are the key to low power system design

Jeremy Bennett, Embecosm

Abstract

Despite a decade of innovative development, and despite improvements in battery technology, a modern smartphone needs recharging far more than its turn of the century predecessor. Yet the blame cannot be laid at the door of hardware engineers. Multiple clock domains, clock gating and dynamic voltage and frequency control have all served to make modern hardware highly power efficient. The problem lies in the software.

In this paper we consider how the entire software design process needs reworking to bring the software engineering team into low power design from day one.

Central to this is the availability of good software development and debug functionality. We look at how the development environment must work seamlessly from the first architectural model to delivery of finished silicon.

The author is one of the main developers of the OpenRISC processor, which has been used to demonstrate some of these ideas. The paper concludes with a short overview of the OpenRISC project.

Low power system design

Has downloading and running the latest applications also drained your smartphone's battery? Consider how technology has advanced over the past decade.

The Ericsson T95 was launched in 2001. It had a 720 mAh Li-Ion battery, a standby time of 300 hours, talk time of 11 hours and a simple indicator of how much standby and talk time remained.

The Sony-Ericsson Xperia X10 mini was launched in 2010. It has a 910 mAh Li-polymer battery, a standby time of approximately 360 hours with 2G and 285 hours with 3G and a talk time of approximately 4 hours with 2G and 3.5 hours with 3G.

The fault cannot be laid at the door of the hardware design team. In recent years, hardware designers have become very good at low power design. Multiple voltage domains, clock gating, dynamic frequency scaling, multiple modes of operation and a host of other techniques have helped reduce power consumption. It is a never ending battle as dimensions shrink to just 10s of atoms, and leakage becomes an ever more pressing problem.

There are three main factors contributing to power loss:

- static leakage—mitigated by reducing voltage;
- dynamic leakage—mitigated by reducing frequency and switching; and
- number of components—mitigated through smaller, simpler silicon and less memory.

Note in particular that reducing voltage is a quadratic gain, and that reducing frequency is a double gain because it also allows voltage to be reduced. With chip voltages ranging from 0.6V to 1.5V, there is the potential of ten-fold gain to be had. This is why it is generally more power efficient to use a multi-core chip running at a lower frequency, rather than a single core.

For the hardware designer, the biggest savings by far are achieved at the architectural level. By the time we reach RTL synthesis, gates and layout, there is little scope for significant power saving. In 2010 LSI Logic and Mentor Graphics summarized this potential in Figure 1.

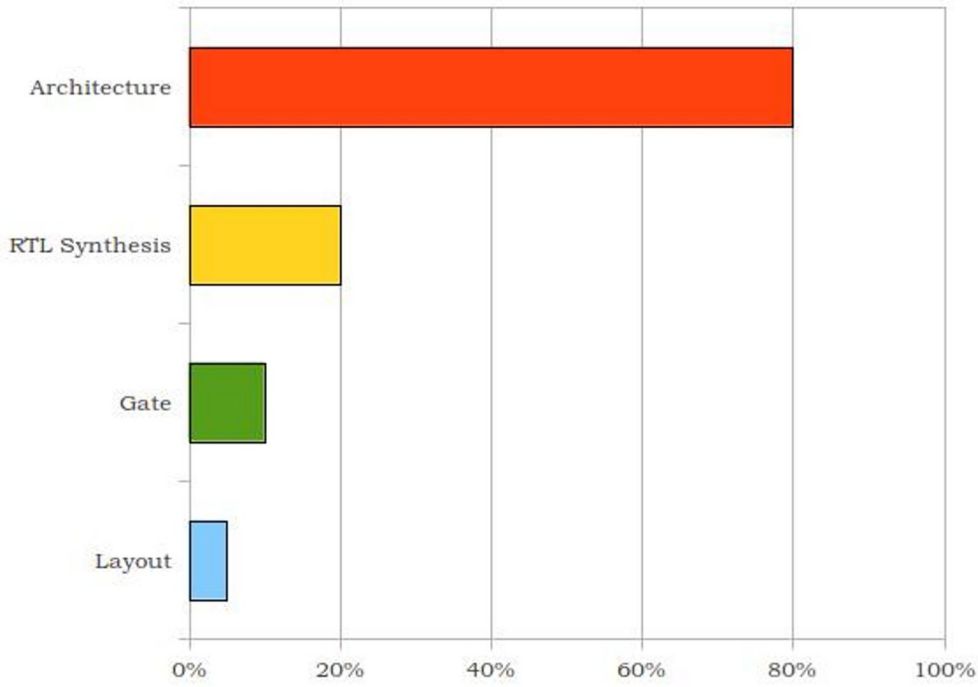

Figure 1: Potential power saving in hardware design

For a long time, energy efficiency has been seen as a hardware problem. Yet software can undo all the design efficiency at a stroke. Famously a Linux implementation wasted 70-90% of its power, simply because a blinking cursor woke up the entire system several times a second [1]. The author was involved in a commercial project, where the design team found they had to increase clock frequency (and hence power consumption) three fold because a standard audio codec caused excessive processor stalls through cache conflicts. That project was canceled shortly afterwards.

Why focus on the system and software in particular? Traditionally, researchers and engineers work within one or perhaps two layers of the system stack with very limited overlap, for example software engineers, computer architects or hardware designers. However, energy-aware computing is a challenge that requires investigating the entire system stack from application software and algorithms, via programming languages, compilers, instruction sets and micro architectures, through to the design and manufacture of the hardware.

Ultimately software controls the hardware. Choice of algorithms and data structures will have a huge impact on power consumption. The traditional compiler focus on speed at the expense of all other considerations is very bad news for power consumption. Few software engineers appreciate this. Power usage is invariably a secondary requirement, if it is a software requirement at all. Yet the biggest savings are to be had at the top of the architectural stack. With Kerstin Eder, the author recently suggested the LSI Logic/Mentor Graphics chart could be extended as shown in Figure 2 [6].

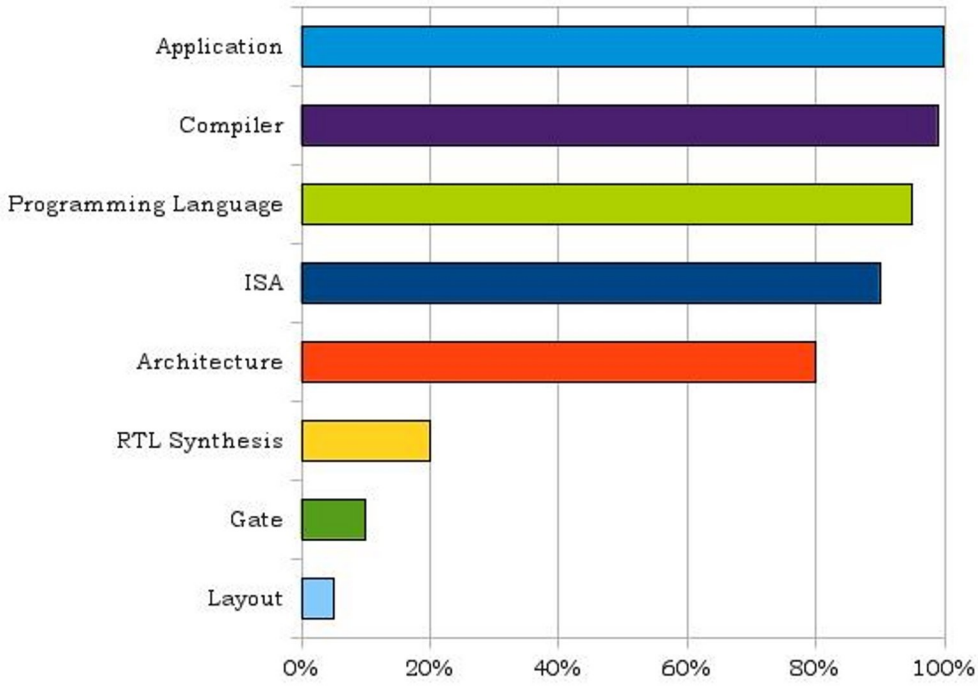

Figure 2: Potential power saving in system design

We do not (yet) have quantitative data to substantiate this chart. Its shape is derived from anecdotal evidence from a number of system designers.

How to tackle energy efficiency at a system level has been known for well over a decade. In their 1997 paper [2], Roy and Johnson summed up how to align software design decisions with energy efficiency as a design goal. Their key steps are (in the given order):

- choose the best algorithm to fit the hardware;
- manage memory size and memory access through algorithm tuning;
- optimize for performance, making best use of parallelism;
- use hardware support for power management; and
- generate code that minimizes switching in the CPU and data path.

One of the reasons for slow progress in this area is the lack of suitable tool flows. Eder [7] has explained exactly what is required. We already know how to do hardware power analysis, as illustrated in Figure 3.

This approach is accurate, but computationally immensely demanding, so the analysis is slow.

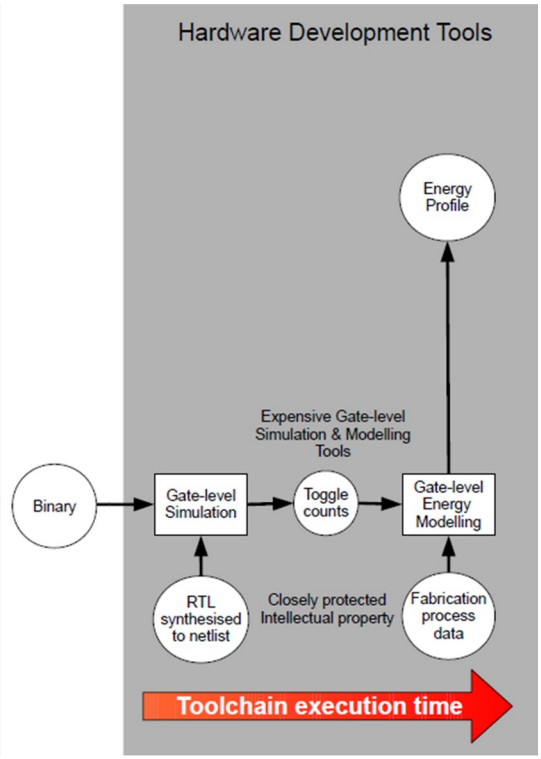

Figure 3: Hardware power analysis flow (from Eder 2011[7])

We can naturally extend this to a system level analysis as shown in Figure 4. However if power analysis of gate level simulation was computationally hard, this approach to power analysis of a complete system including software and hardware is completely intractable.

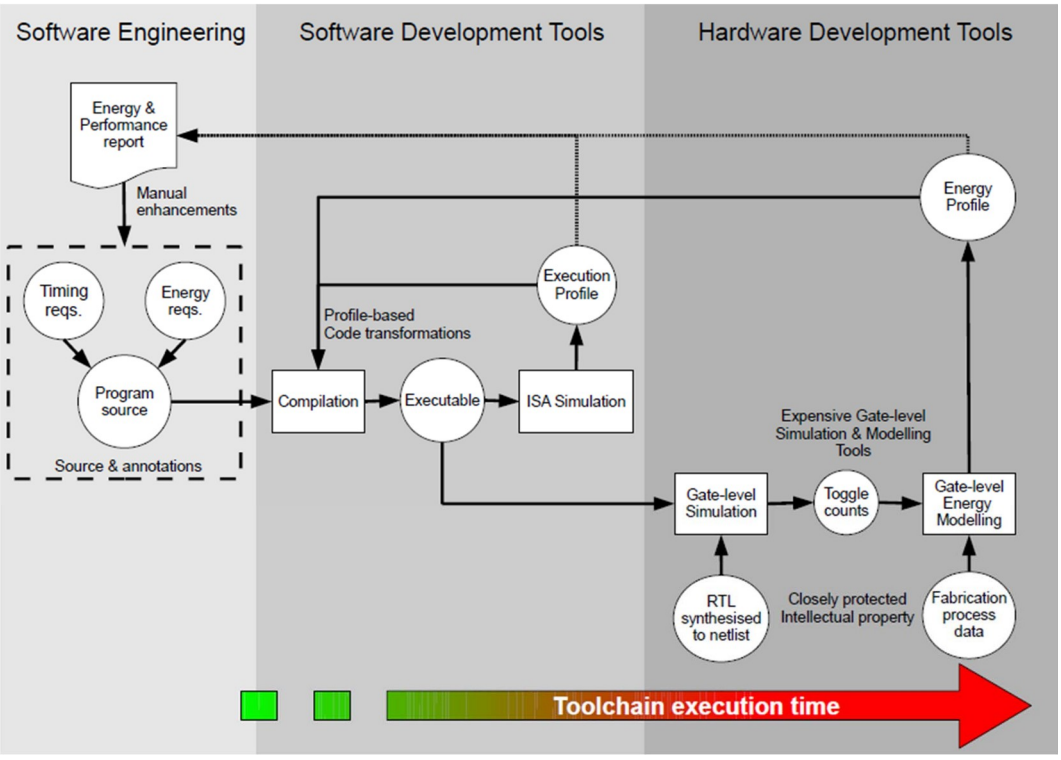

Figure 4: System power analysis flow (from Eder 2011 [7])

What is needed is power analysis appropriate to the needs of system and software development. In other words a flow like that in Figure 5.

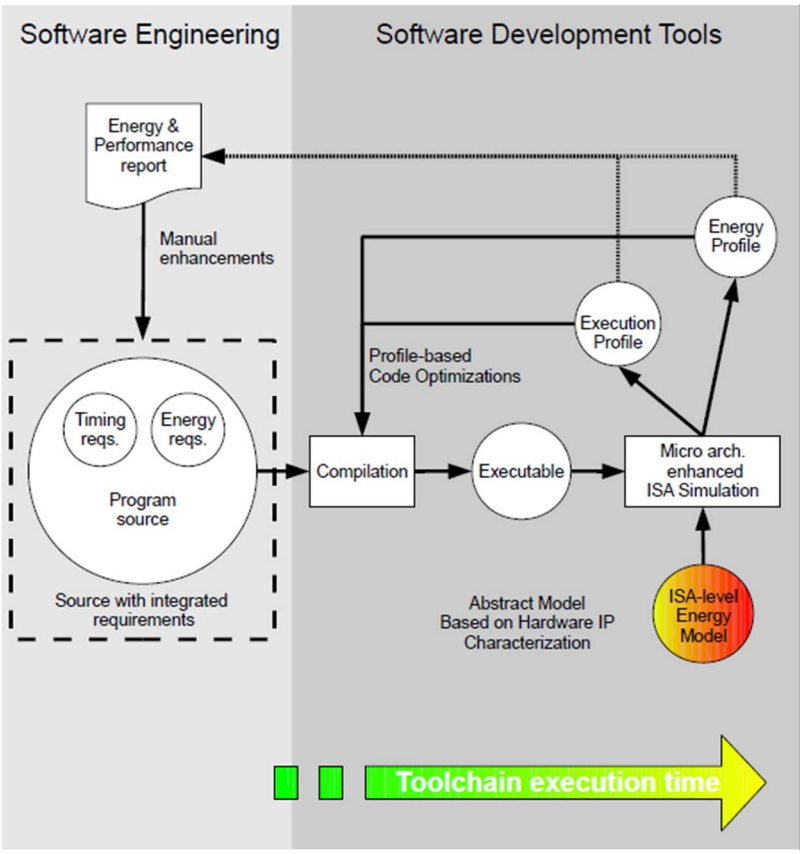

Figure 5: Software power analysis flow (from Eder 2011 [7])

The problem is in modeling power consumption, even when we are prepared to tolerate a degree of inaccuracy. This is an area where progress has been slow over the past 15 years.

An early approach was to use a formulaic analysis based on the operations in executing code [8].

$$E_P = \sum_i B_i \times N_i + \sum_{i,j} O_{i,j} \times N_{i,j} + \sum_k E_k$$

The formula contains a term for the base power used by each instruction, a term for the power overhead in switching between each pair of instructions and a term for various other instruction effects such as pipeline stall. The formula is highly parameterized, and determining the values of those hundreds of parameters experimentally or from first principles is difficult. Yet without accurate parameters, the results cannot be accurate.

Wattch is an architectural level power simulator, which instead estimates system power usage by combining common functional blocks, whose power usage is already determined [3]. This is a practical approach, which is reported to offer accuracy within 10% and a performance one thousand times greater than traditional gate level power estimation.

Using these approaches we have learned some things about how to design low power systems.

One study minimized the Hamming distance between pairs of instructions to reduce switching [9]. This reduced power consumption by 62% in opcode switching. The problem with this approach is that it yields an ISA which is very target application specific.

Another study found that 25% of the registers in a register file accounted for 83% of the time spent accessing the register file. It thus makes sense to partition a register file into

"hot" and "cold" register blocks. This led to a 54% reduction in power consumption by the register file compared to an unpartitioned register file [10].

Other researchers have looked at higher levels of abstraction still. One approach is use of approximate calculation to reduce the computation required, where full accuracy is not required [4]. A related approach allowed the programmer to control the number of bits of accuracy used in floating point applications [5].

However both these studies must be regarded as "niche", and of little relevance to software engineering in general.

Perhaps one of the best approaches is one of the simplest. Measure the power being consumed as code executes on a chip. This can be as simple as measuring the voltage drop across a resister in the power line [11,12]. The resulting data can then be reconciled with program execution to yield an instruction by instruction power profile.

More research is needed in this field. The Energy Aware Computing (EACO) initiative began with 3 workshops during 2011. Sponsored by the Institute for Advanced Study at Bristol University, it aims to foster a European program of research in this general area. Both incremental improvements and radical new innovative approaches are sought. The conveners are Prof David May and Dr Kerstin Eder, both at Bristol University and the next workshop takes place on 18 April 2012 in Bristol.

A system wide approach to debugging software

As we have seen successful low power design relies on having tools that can take a system-wide view. One area where this can have most effect is in debugging software. The traditional approach to system development and debugging is shown in Figure .

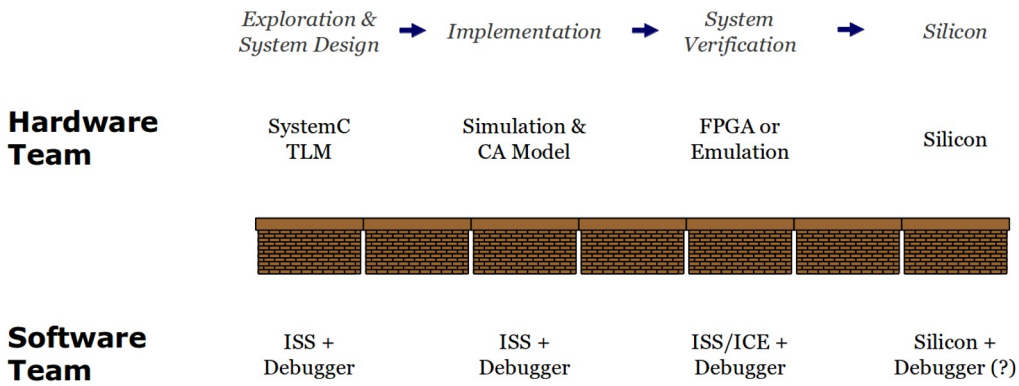

Figure 6: Traditional system development

All too often the hardware and software teams do not communicate. They may be in a different building, different town, different state or even different country and time zone. The software engineers rely on their own ISS, often until after tape out. If only they could use the same models as the hardware engineers.

Embedded software tools such as debuggers that take a system-centric view need two characteristics.

1. They need to be peripheral aware. When the program halts, the peripherals must also halt and the tool must have visibility into peripheral state.
2. They must work with hardware models as easily as with final silicon. That is models of the complete system, not just the CPU, whether high level or low level, software or FPGA emulation.

This is not a technical challenge for the tool developer. Most debuggers are easily extensible to access peripherals, IEEE 1159.1 JTAG (or its successors) provide a natural point of abstraction for the interface and the EDA world knows how to model complete systems.

The GNU debugger (GDB) for the OpenRISC 1200 Reference Platform System-on-Chip (ORPSoC) already supports access to peripheral state. Memory mapped peripheral registers appear as *special purpose registers* (SPRs). GDB is easily extended to add a command to read SPRs, for example to read the programmable interrupt controller (PIC) match register.

```
(gdb) info spr picmr
PIC.PICMR = SPR9_0 = 0 (0x0)
(gdb)
```

Similarly GDB can write the value of peripheral registers.

```
(gdb) set spr picmr 0x00000007
PIC.PICMR (SPR9_0) set to 7 (0x7), was: 0 (0x0)
(gdb)
```

All that is required is a command to provide control based on peripheral registers.

```
(gdb) pwatch picsr
Peripheral watchpoint 2: PIC.PICSR (SPR9_2)
(gdb)
```

This is yet to be implemented, since it requires extension to the underlying hardware.

In an embedded environment the debugger client must communicate with the target, and these additional peripheral commands must be transferred to the target. Each debugger has its own communication protocol, which must be utilized for this purpose, and typically offers appropriate extension facilities.

In the case of GDB, communication is through the GDB *remote serial protocol* (RSP). This simple packet protocol provides one packet, **`qCmd`**, for the express purpose of passing arbitrary commands to the target. In this case we add **`readspr`** and **`writespr`** commands and extend the RSP server on the target to read and write peripheral state according to these commands. By working through this standard interface, these extensions are robust to changes in future GDB upgrades.

There are many types of target model, all with their own interfaces. A SystemC transaction level model (TLM) may define a class with a transactional port as interface.

```
class SocTlmModel
  : public sc_core::sc_module
{
  …
  tlm:tlm_transport_dbg_if<JtagPayload> jtagPort;
```

A SystemC cycle accurate model may define a class with sc_in and sc_out wires as interface.

```
class SocCycleModel
  : public sc_core::sc_module
{
  …
  sc_in<bool> jtagTck;
  sc_in<bool> jtagTms;
```

An FPGA model may interface through library calls to control a JTAG driver chip:

```
static void
jp1_ll_reset_jp1()
{
  …
  write (lp, &data, sizeof (data));
```

```
JP1_WAIT ();
```

We need a mechanism where the debugger interface does not need to be rewritten for each variant. As shown in Figure 7, the solution is to use a transaction level abstraction of JTAG as the interface between debugger and target [13].

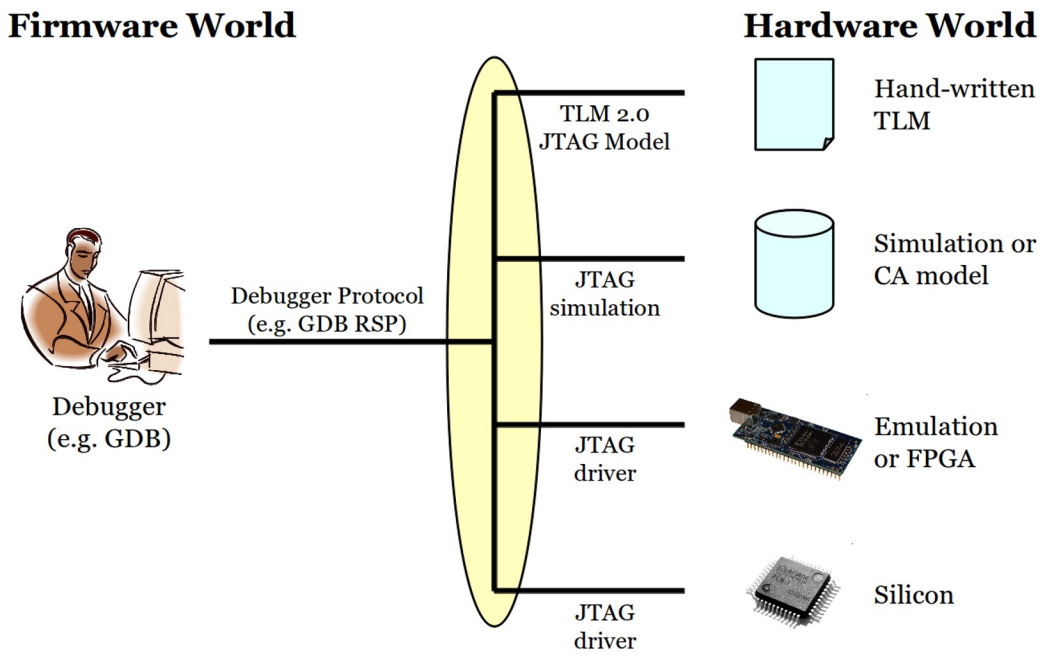

Figure 7: Unified debug using JTAG as interface abstraction

At its simplest, JTAG is just a mechanism to write and read serial registers of arbitrary size, so naturally fits a transactional model of abstraction. The highest level of class abstraction is of the RSP server class communicating with a JTAG interface class mediated via a JTAG register class as shown in Figure 8.

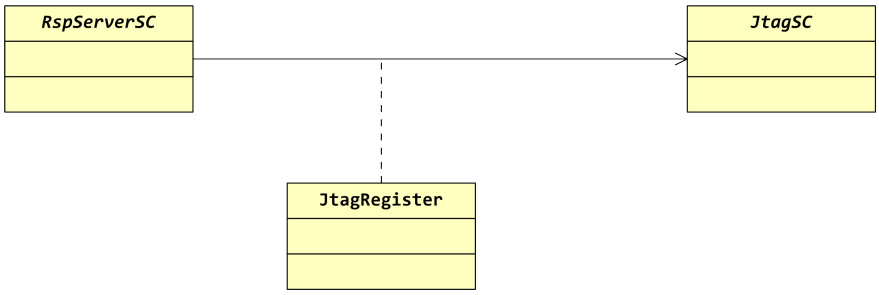

Figure 8: High level class abstraction for unified debug interface

All three classes are abstract. Of each interface we must provide a concrete specialization corresponding to the particular interface as shown in Figure 9.

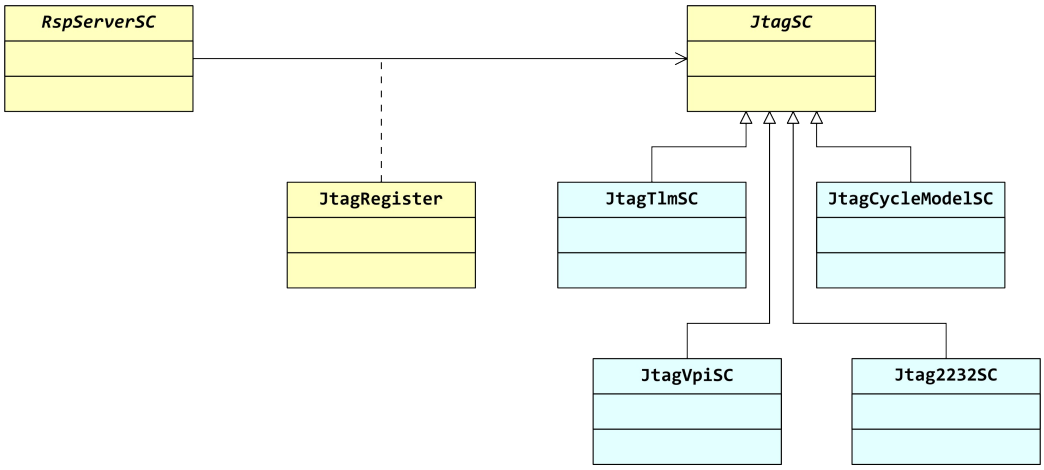

Figure 9: JTAG interface specialization for unified debug interface

Note that these specializations are independent of any particular architecture being debugged.

For any particular architecture we must then provide specialization of the RSP server class and JTAG class. These will provide handling of architecture specific debug packets. For example translating the reading and writing of memory into the correct sequence of JTAG packet transfers. We must also provide the specific JTAG registers used by the processor's particular debug unit. This gives us the class specialization shown in Figure 10.

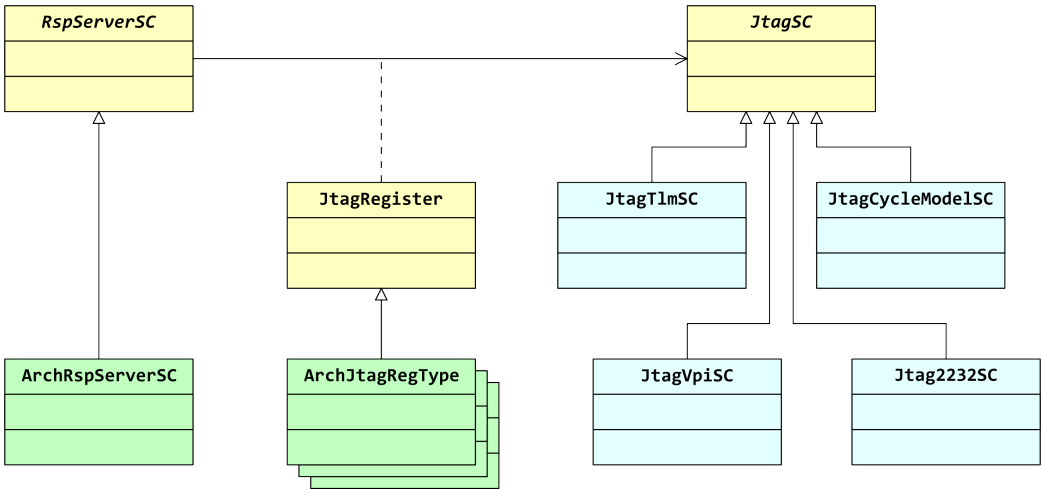

Figure 10: RSP interface specialization for a unified debug interface

Note how this approach allows efficient reuse of the interface. For any new architecture, providing just the architecture specialization allows the debugger to talk to all types of models and hardware for which a JTAG specialization exists. Similarly for a new type of debug interface (for example to a different JTAG chip), it is only necessary to provide a new JTAG class specialization and that interface is available to any architecture.

The SystemC transaction level modeling interface cannot be used in its vanilla form, since it does presume a byte addressed bus interface. It also assumes read and write are separate operations and has no concept of the simultaneous write and read of a shift register. However the TLM extension mechanism allows this functionality to be provided in a

standard way, ensuring models will work in any TLM environment. The extension class diagram is shown in Figure 11.

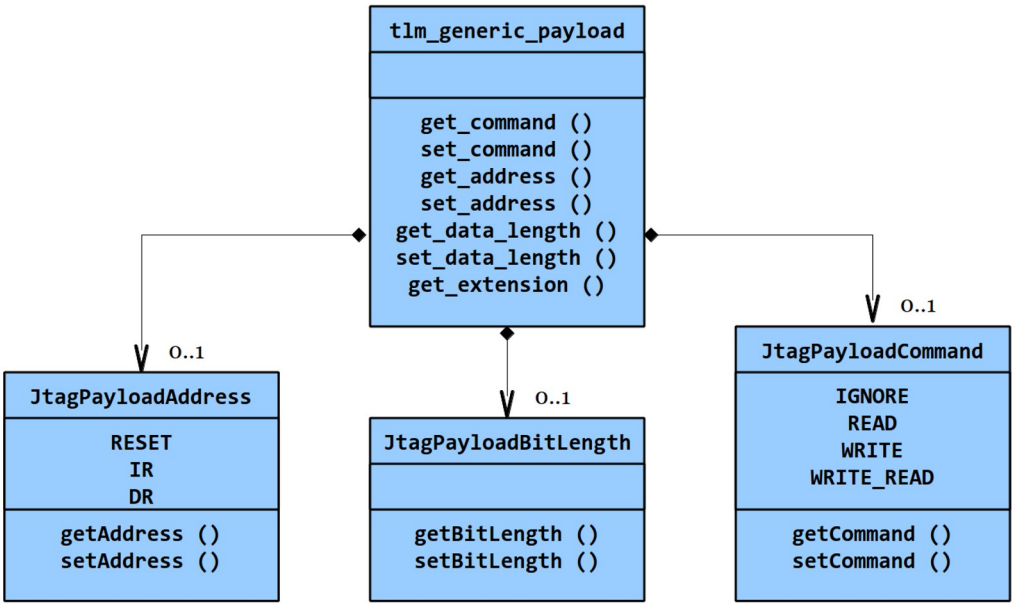

Figure 11: SystemC TLM extension classes for JTAG

The specialization of this abstraction (which has no time component) to lower level interface will require modeling of the JTAG test access port (TAP) to generate the correct sequence of clock signals on the JTAG pins. Figure 12 shows how this works as a sequence diagram.

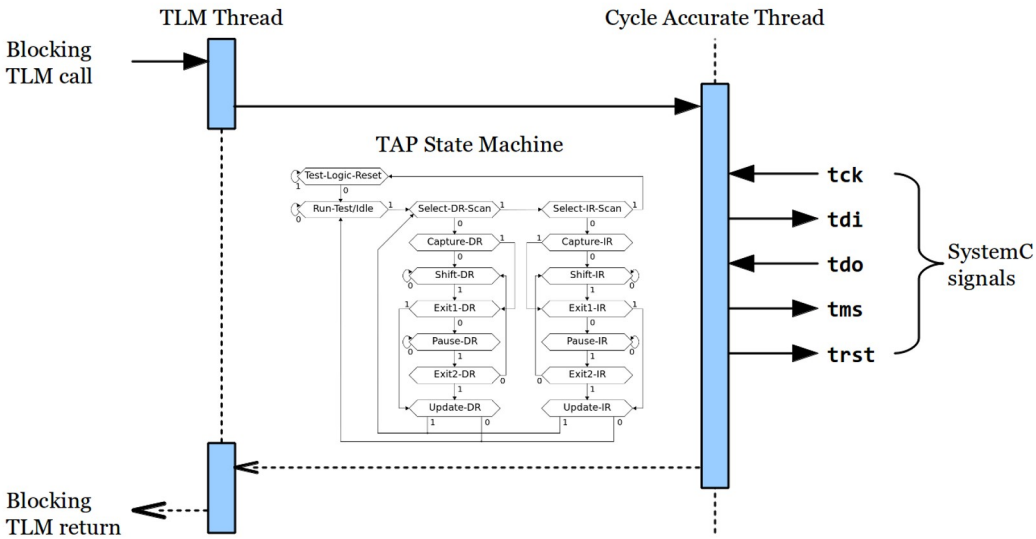

Figure 12: JTAG TLM to cycle accurate specialization

OpenCores and OpenRISC

Much of the work described in the previous section has been demonstrated on the OpenRISC processor. We provide an introduction here to OpenRISC and the OpenCores website on which it is hosted.

OpenCores was conceived as a website to host open source silicon IP in 1999 by Damjan Lampret, then at Flextronics. While the intention was always to host a wide range of IP, from

the beginning its flagship project was the development of a fully open source RISC processor.

For the first eight years, the development of opencores.org was supported by Flextronics, with most contributors being coming from their European development teams. The result included a reference ASIC implementation with approximately 150,000 gates and 17 memory blocks.

Since 2007, opencores.org has been run by ORSoC AB, a Swedish hardware design house. Under their stewardship the website has grown to around 120,000 registered users (at the time or writing). Discussions are under way now to set up a completely independent foundation, with the community now mature enough to take control of its own destiny.

The OpenRISC processor has been adopted in a number of commercial applications. Beyond Semiconductor is a design house, supplying commercially hardened derivatives of the OpenRISC processor. Jennic (now part of NXP) was an early adopter of the Beyond Semiconductor designs for their Zigbee chips. Samsung use OpenRISC in their DTV SoCs. Cadence use OpenRISC as a reference architecture to demonstrate their various EDA design flows.

Most OpenCores IP designed are licensed using either BSD or Gnu LGPL licenses (OpenRISC uses the latter). Associated software tool chains and documentation are typically licensed using the Gnu GPL.

LGPL does represent something of a problem for silicon IP, even IP that is destined to become a bitstream on an FPGA. What is the hardware equivalent of linking? Is an FPGA bitstream equivalent to a software object file?

The OpenRISC 1000 architecture defines a family of 32 and 64-bit RISC processors with a Harvard or Stanford architecture [14]. The instruction set architecture (ISA) is similar to that of MIPS or DLX, offering 32 general purpose registers, register-to-register operations, two addressing modes, delayed branches and a fast context switch. The processor offers WishBone bus interfaces for instruction and data 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 13.

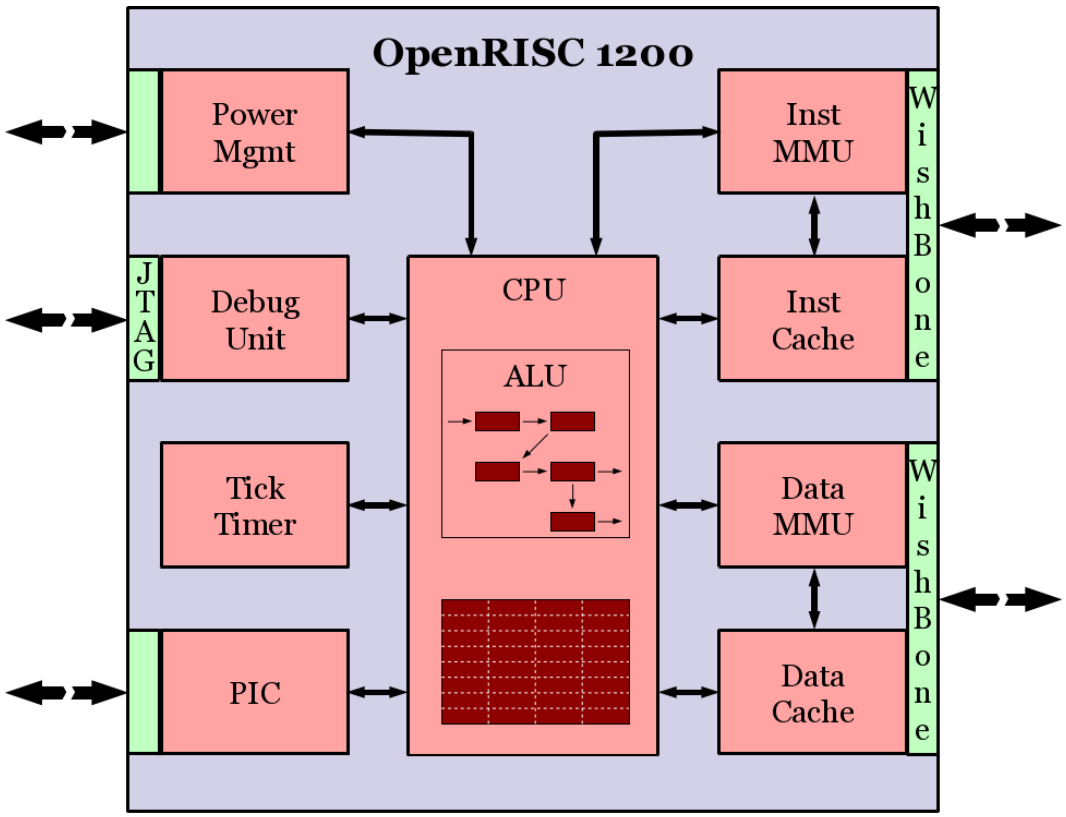

Figure 13: OpenRISC 1200 architecture

For development and testing purposes the OR1200 is incorporated into a System-on-Chip (SoC), the OpenRISC Reference Platform SoC (ORPSoC), which is shown in Figure 14.

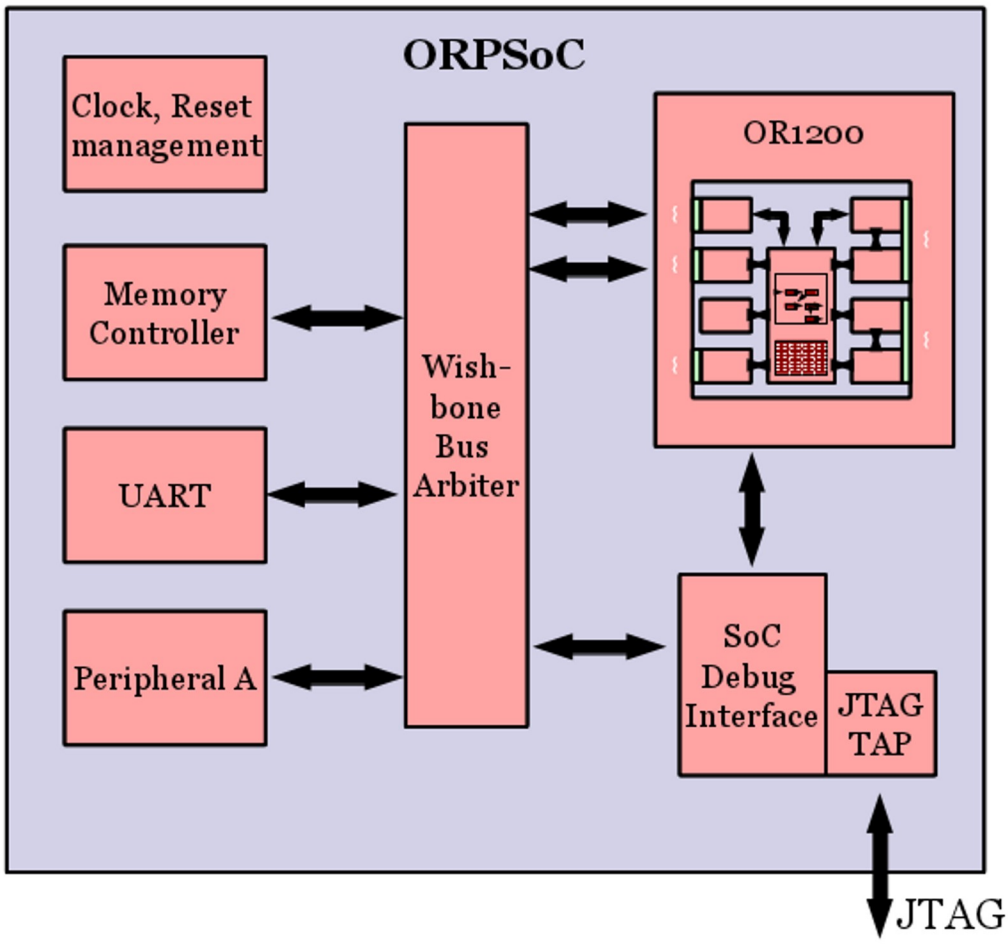

Figure 14: OpenRISC Reference Platform SoC (ORPSoC)

One objective of the OpenRISC project is to use open source tools as far as possible during hardware development. While back-end FPGA tools are free (as in beer), the are not open source. However there are now good open source options for front-end EDA tools.

There are three simulation models used during OR1200 development

- Or1ksim, the golden reference architectural model. This is an interpreting ISS running at 2-5 MIPS with a test-suite of approximately 2,500 tests.
- A 2-state cycle-accurate model in C++/SystemC generated automatically from the Verilog RTL by Verilator, which runs at around 130kHz.
- An event driven simulation model of the Verilog RTL using Icarus Verilog, which runs at round 1.4kHz.

All three models support the GDB RSP, so can be used for software development and debug. This is key to a system-centric view of the development process.

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. The test bench comprises a total of 13 target programs, which were deemed to past if they printed out 0xdeadbeef on exit.

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.

More recently an OVM testing regime for the OR1200 was implemented [15]. 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 15.

Constrained random test generation was used to create a set of tests to maximize coverage. Testing uncovered numerous errors where the RTL and Or1ksim disagreed, which fell into three categories.

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.

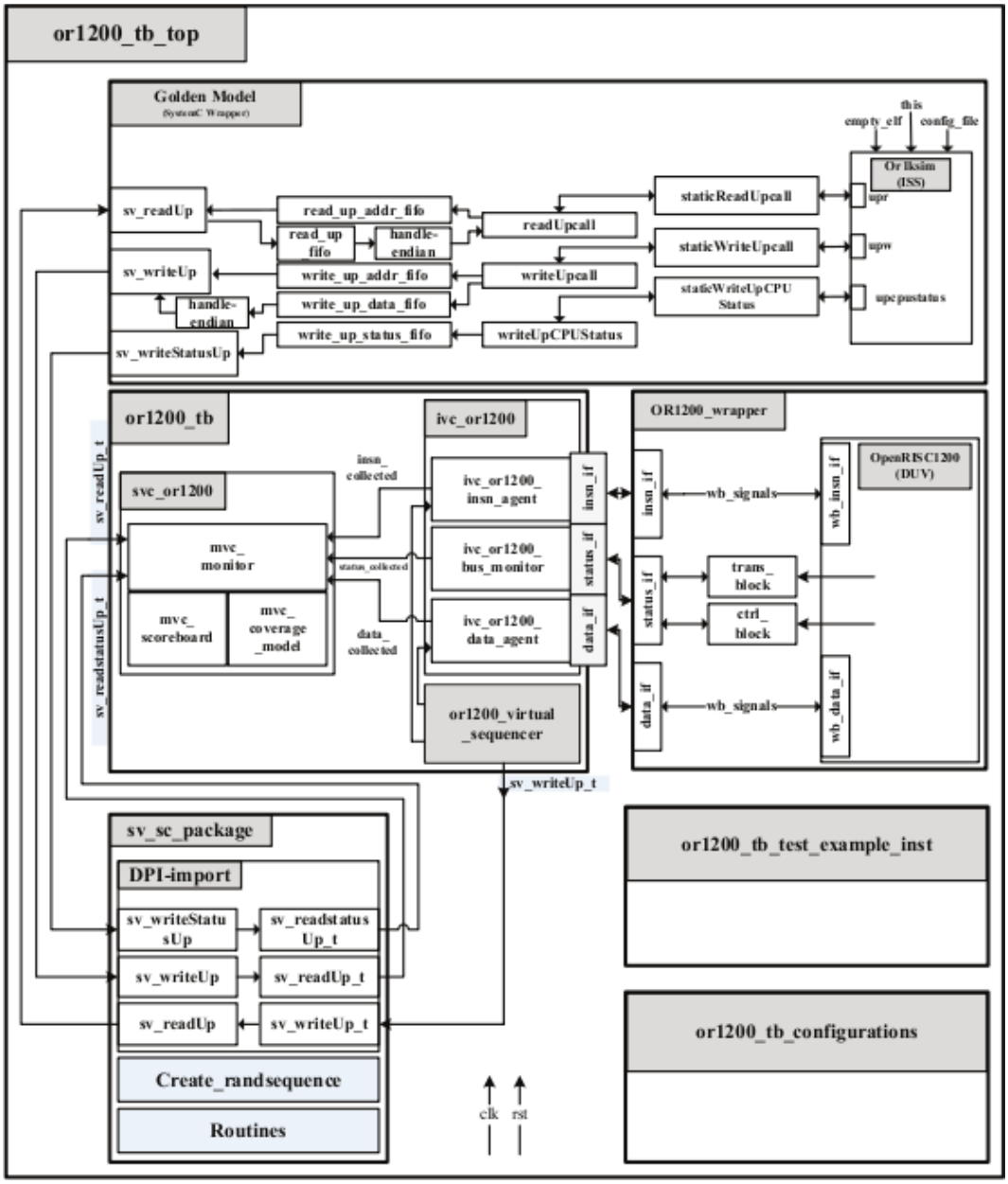

Figure 15: Dual Or1ksim and RTL test bench under OVM

The OpenRISC 1000 architecture is supported by a comprehensive tool chain. *binutils* 2.20.1, gcc 4.5.1 and gdb 7.2 are implemented supporting both C and C++. At present, in common with many embedded tool chains, only static libraries are supported. The **`or32-elf-`** tool chain for bare metal applications uses the *newlib* library, while the **`or32-linux-`** tool chain for Linux applications uses the *uClibc* library. Both tool chains are regression tested on Or1ksim, with the **`or32-elf-`** tool chain also tested against the Verilator model and the **`or32-linux-`** tool chain tested on physical hardware.

A wide range of boards have board support packages, including Or1ksim, the Terasic DE0-nano and DE2 FPGA boards and the Xilinx ML510 FPGA board.

A number of open source real-time operating systems (RTOS) are supported. FreeRTOS, RTEMS and eCos have all been ported to OpenRISC.

OpenRISC supports Linux, with the implementation being adopted in the Linux 3.1 mainline. There are currently some limitations to the implementation (kernel debug, ptrace). BusyBox is supported as a command-line environment.

Debug is supported through JTAG, which is appropriate for bare-metal applications and the Linux kernel. For Linux applications, *gdbserver* has been implemented.

The ability to seamlessly use the tool chains on both models and physical hardware leads to a new way to carry out system verification of the hardware [16]. The GNU tools all contain a very substantial regression test-suite (around 100,000 tests). Comparative runs against the reference architectural model, Or1ksim and the Verilog implementation (on physical hardware or as Verilator model) should give identical results.

Any discrepancies can be down to one of two reasons:

1. tests timing out on one target, due to differential performance; or
2. bugs in the hardware implementation.

As an example, early in the implementation of GCC 4.5.1, we were able to run the gcc regression tests on both targets. On Or1ksim, the results were:

```
=== gcc Summary ===

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

With a Verilator model of the RTL, the results were:

```
=== gcc Summary ===

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

The 76 tests which failed with the Verilator model were then examined to determine the cause. 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 another example of the second type of failure.

There is another 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.

These bugs have now been corrected in the OpenRISC RTL. The compiler implementation has also been completed, and there are now no regression failures with either Or1ksim or the Verilog RTL implementation.

This approach has been used commercially. For example Embecosm developed the GNU tool chain for the Adapteva Epiphany architecture prior to first silicon tape out. Adapteva reported that the discrepancies found enabled the elimination of 50-60 hardware design flaws, leading to a processor that worked correctly with first silicon.

Summary

In summary

- Future low power products will require a systems approach. This will mean hardware and software engineers must work together and the approach applies throughout the life cycle.
- The greatest opportunity for power saving is in the software. Techniques for tackling this are still in their infancy. We need breakthroughs in high level power modeling and simulation
- We need a systems oriented tool chain geared to the needs of both software and hardware and usable throughout the product life cycle
- Embecosm's unified debugging approach is an example, which allows software debugging throughout the life cycle
- The benefits can be seen already in the OpenRISC project, with hardware bugs identified by the software engineers

Acknowledgments

Most of the work described in the first section of this paper is due to my colleague, Dr Kerstin Eder at the University of Bristol. It draws heavily on our joint paper for the NMI [6].

OpenRISC is a community project, to which I am just one of the contributors. It is the cumulative result of 12 years work by a very large number of people.

References

1 E. Sperling and P. Chatterjee. 16 June, 2011. The Tao of Software. *Chip Design Magazine Low Power Engineering* online community blog post. chipdesignmag.com/lpd/blog/2011/06/16/the-tao-of-software.

2 K. Roy and M.C. Johnson. 1997. Software design for low power. In W. Nebel and J. Mermet (Eds.) *Low power design in deep submicron electronics*. Kluwer Nato Advanced Science Institutes Series, Vol. 337, Norwell, MA, USA, pp 433-460.

3 D. Brooks, V. Tiwari, M. Martonosi, *Wattch: A framework for architecture-level power analysis and optimizations*, Proc. 27th International Symposium on Computer Architecture (ISCA), pp. 83-94, 2000.

4 A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze and D. Grossman, *EnerJ: Approximate Data Types for Safe and General Low-Power Computation.* In Proc. of PLDI, June 2011.

5 J.Y.F. Tong, D. Nagle and R.A. Rutenbar, Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic. *IEEE Transactions on VLSI Systems*, 8(3), pp 273-286, June 2000.

6 Jeremy Bennett and Kerstin Eder. *The software drained my battery.* NMI yearbook 2011-12, pp 39-24, November 2011.

7 Kerstin Eder. *Energy aware system design.* Low-Power Verification - bridging the gap between hardware and software. NMI meeting, Rutherford-Appleton Laboratory, 20 June 2011.

8 V. Tiwari, S. Malik and A. Wolfe, *Instruction Level Power Analysis and Optimization of Software*", Journal of VLSI Signal Processing Systems, 13, pp 223-238, 1996.

9 S. Woo, J. Yoon and J. Kim, *Low-Power Instruction Encoding Techniques*, Proceedings of the SOC Design Conference, 2001.

10 X. Guan and Y. Fei, *Registeer File Partitioning and Recompilation for Register File Power Reduction, ACM Transactions on Design Automation of Electronic Systems, 15(3)24, May 2010.*

11 Steve Kerrison, Bristol University Design Automation and Verification, Microelectronics Group. Informal presentation during the 2nd EACO Workshop, *Energy-Aware Computing: Beyond the state of the art,* 13-14 July 2011.

12 Barry Lock, *Power by Software Function - some real life examples*. Low-Power Verification - bridging the gap between hardware and software. NMI meeting, Rutherford-Appleton Laboratory, 20 June 2011.

13 Jeremy Bennett, *Using JTAG with SystemC: Implementation of a Cycle Accurate Interface*, Embecosm Application Note No. 5, www.embecosm.com, January 2009.

14 The OpenRISC project. http://opencores.org/or1k/Main_Page.

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

16 Jeremy Bennett, *Processor Verification using Open Source Tools and the GCC Regression Test Suite: A Case Study,* Design Verification Club meeting, 20 September 2010, Bristol UK, Cambridge UK and Eindhoven Netherlands (multicast conference).

About the Author

Dr Jeremy Bennett is Chief Executive of Embecosm (www.embecosm.com) which provides open source services, tools and models to facilitate embedded software development with complex systems-on-chip. Contact him at jeremy.bennett@embecosm.com.

This paper was presented to the School of Electronic, Electrical and Systems Engineering on 14 December 2011.