Platform based on Open-Source Cores for Industrial Applications

M. Bolado[1], H. Posadas[1], J. Castillo[1], P. Huerta[1], P. Sánchez[1], C. Sánchez[2], H. Fouren[2], F. Blasco[2]

[1]*University of Cantabria. Microelectronics Engineering Group. TEISA. Santander. Spain.*

[2]*Design of System on Silicon (DS2). Paterna. Valencia. Spain.*

Abstract

The latest version of the International Technology Roadmap for Semiconductors predicts that design reuse will be essential in the near future to face the constantly increasing design complexity. The concept comes from software engineering in which reuse is a fundamental technology. In order to provide libraries and applications to reuse in software development, some open-source initiatives (e.g. Linux, gcc, X, mysql) have appeared during the last decades. The basic idea is to distribute the library or application source code (normally free-of-charge) and allow any developer to use, modify, debug and improve it. Several initiatives have tried to port this idea to hardware development. The main goal of this paper is to develop a synthesizable platform described in SystemC from an open architecture. The platform includes a CPU (OpenRISC) and some basic peripherals. A set of software development tools (compiler, assembler, debugger) and RTOS (eCos) has also been developed. This work enables the evaluation of the advantages and disadvantages of the open-source model in electronic system design.

1. Introduction

In order to allow the huge increase in design productivity that seems necessary to exploit the constantly increasing system and silicon complexities, a system level design methodology that allows reuse-based and platform-based design in both HW and SW domains will be essential in the coming years [1]. This has created a new business segment (commerce in HW and SW IPs) in which a lot of IP-developments, IP-vendors and IP-catalogs have appeared during recent years. However, to be practical, the reuse-based methodology must guarantee that the IP integration process is successful (satisfying specification and constraints, error-free and cheaper than homemade development) thus some proposals have been made with this objective. Firstly, some standards (e.g. VSIA standards [2]), specification languages and IP design rules (e.g. Reuse Methodology Manual [3]) were defined. Secondly, some electronics catalogs that facilitate core selection and transfer were developed (e.g. [4][5]). Finally, some CAD tools that provide the necessary infrastructure for IP-based design were proposed [6].

But even taking into account the previously commented techniques, reuse can doom a project to failure. This has forced a review of approaches (e.g. VSIA [7]) and an analysis of the main cost involved in reuse [8]. There are three primary metrics that can determine the magnitude of cost and saving via reuse: original development time, amount of design modification and verification effort. Verification is one of the main bottlenecks of system level design, thus it is also a problem in IP-based design [9].

Another problem is IP modifications. In theory, only the IP interface can be modified, but in practical cases some modifications have to be introduced in the IP to cover specifications and constraints. A core is not really reusable until it has been reused (and modified) several times [10]. Additionally, it is expensive to do forward-looking design of a function or module; today it is easer and cheaper to solve very specific problems than anticipate demands of future projects. Thus, new projects sometimes require new features of existing cores that have to be implemented in the IP. The core provider can do these modifications (commercial solution) with a substantial increment of the core cost. Another possibility (ad-hoc solution) is to use open-source cores in order to create an internally developed core [8].

The open-source approach seems to have several advantages: the core is very cheap (normally free), the user can have source code access and there is a group of developers that provide know-how, maintain and improve the core. However, it may also have several disadvantages such as instability (the development group changes or

This work has been supported by the MEDEA+ A511 ToolIP project and the Spanish MCYT through the TIC-2002-00660 project.

1530-1591/04 $20.00 (c) 2004 IEEE

disappears), incomplete development, poor or no support of existing IP-reuse infrastructures and standards, poor documentation and verification methodology.

The main goal of this paper is to explore the ad-hoc solution to enable reuse. Thus, a microprocessor (based on the open-source OpenRISC core [11]) and the basic HW (busses, memories, peripherals) and SW (compiler, debugger, RTOS) platform elements have been developed. The system has been described in SystemC and implemented within a FPGA.

The quality of the open-source core will be analyzed in the next section. After this, the developed platform will be presented (section 3) and its main hardware (section 4) and software (section 5) components described. Section 6 will comment the core verification methodology and environment. Finally, the simulation and synthesis results will be presented on section 7 and some conclusion will be provided on section 8.

2. Open-Source IP Core Quality

IP core quality assessment is an important issue in reuse-based design methodologies. Many metrics and techniques have been proposed for this objective, such as the VSIA Quality IP Metric (QIP)[2] or the Mentor&Synopsys OpenMORE [21]. In this work, we have used the OpenMORE quality assessment program to evaluate the open-source IP core. We have selected this program because it can be downloaded free, it has been used to qualify some commercial cores and it was donated to VSIA and integrated into QIP (currently under VSIA member review).

The first step is to analyze the core distribution. The OpenRISC Team at OpenCores [11] has developed a first implementation: OpenRISC 1200 (OR1200). This soft core is a MIPS-based 32-bit scalar RISC with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. The core has been described in Verilog, verified with several functional tests and implemented into FPGAs and ASICs. The distribution also includes a complete Software Development Kit (SDK) based on GNU tools. It includes binary utilities (assembler, linker), C/C++ compiler, debugger and an architectural simulator. There is also a port of the μClinux Operating System [13] and some groups are working to port other OS such as Linux, RTEMS [12] and eCos [19]. The OpenRISC Team has also developed a platform specification (OpenRISC Reference Platform, ORP) and a platform example (ORPSoC) that includes RTOS (μClinux) and bootstrapping monitor (ORPmon). There are also development boards and silicon implementations of this platform[11].

When the selected quality assessment program (openMORE) is applied to the previously described IP core, the first problem is that only the synthesizable RT model of the core is evaluated and the rest of the distribution (basically, the SDK) is ignored. OpenMORE splits the soft-core evaluation into 3 main sections:

- **Macro Design Guidelines**. The OpenCores project provides a HDL coding guideline document [22] that verifies most of the OpenMORE recommendations. Additionally, the main core developers are design-company engineers that use standard industrial design practices. Thus, the overall coding-style quality of the OR1200 core is quite good (283 of 396 points, that is an OpenMORE rating of 71%). The Macro Design Guidelines have three sections: System-Level Design Issues, RTL Coding and Synthesis Guidelines. Concerning "System Level Design Issues", the core rating is very good (64 of 70 points, rating of 91%). The weaker aspect of the core is the documentation of clocks. The rating of the RTL coding section is lower (158 of 218 points, a 71%), mainly because the naming and port conventions are different. The poorest rating is obtained in the synthesis section (61 of 108 points, 51 %) because the distribution only includes a simple global synthesis script.

- **Verification Guidelines**. The OpenCores IPs should fulfill the verification strategies defined in [23]. This draft is a preliminary version that defines the main verification procedures but it is poorer than up-to-date approaches such as the VSIA functional verification deliverables [17]. Nevertheless, the verification rating of the open-source core is quite good (54 of 74 points, 73%). The rating of the macro verification section is very poor (11 of 22 points, 50%) but the system level verification rating is very good (43 of 48 points, 89%). These results were surprising because it was thought that verification was one of the weaker aspects of the core. The reason could be the verification goals. The main goal of the OR1200 verification environment might be to enable the IP integrator to perform an acceptance test or even and integration test. However, in our work we needed a more complete verification environment that could validate source code modifications.

- **Deliverable Guidelines**. The OpenRISC deliverables obtain a good rating: 74 of 104 points that is a 71.4%. The weaker part is related with the "verification files".

As conclusion, the overall quality of the open-source core is 411 of 570 points that is a rating of 73%. In OpenMORE, a rating greater than 60% usually means that the core is "good for reuse" thus we can conclude that the OpenRISC core has enough quality to be integrated in an industrial development.

3. Proposed OpenRISC Based Platform

One of the requirements of this work was to develop in SystemC a platform that could be included in a low cost and high performance family of products. An OR1200-based platform does not verify this requirement thus a new platform had to be developed. It includes a new microprocessor core (OpenRISC 1500) and the minimum set of elements needed to provide basic functionality (see Figure 1). Additionally, the platform should include an operating system with low memory-size requirements such as the open-source eCos RTOS.

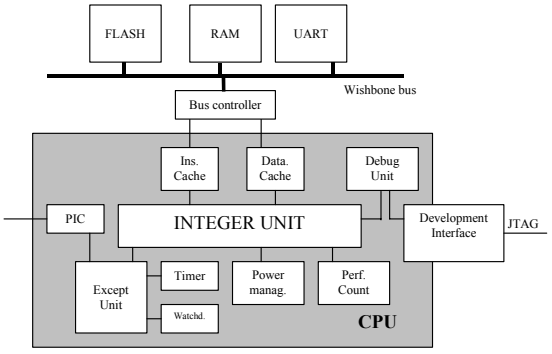

Figure 1: Proposed Platform Structure

The basic communication channel of the platform is an OpenCores Wishbone Compatible Bus [14]. The bus master controller connects the CPU with the peripherals. Additionally, it provides peripheral access for in-circuit debugging. The platform also includes the slave controllers that manage the access to Flash and RAM memories and, optionally, it could include a 16550 compatible UART. The synthesizable description of the CPU peripherals (excluding the UART) takes about 2500 SystemC code lines.

4. CPU Description

OR1500 is a 32-bit RISC processor that fulfils the OR1000 architecture and offers a lot of configuration possibilities and optional units that enable its use in a wide range of application. It is composed of eleven units (see figure 1) that will be next described in order to provide a system complexity overview. These units have been described in about 14500 SystemC code lines.

4.1. Integer Unit

The pipeline of this unit has been optimized in order to improve the OR1200 performance. Several simulations of the proposed Integer Unit have shown that it takes about 17% fewer cycles than the OpenRISC 1200 implementation. The SystemC description of this unit has 4100 code lines.

4.2. Data and instruction caches

The degree of associativity was decided after several test-case simulations. The decision was to implement 2-way associative caches. Cache size and data block size are parametrizable. Cache size can be set from 2 KB to 8 KB and the block size can be set to 16, 32 or 64 bytes. The LRU (Least Recently Used) replacement algorithm is used. The system integrator can choose between the two possible cache write policies: copy back and write through [15]. Finally, two additional techniques are used to improve performance: load through and write buffer [15]. The data and instruction caches are described in 5900 SystemC code lines.

4.3. Debug unit and development interface

The debug unit is an optional facility that provides the ability to create hardware breakpoints and watchpoints based on complex comparison conditions with stored or loaded values, data and instruction memory addresses. It is very closely related to the development interface that allows complete in-system debugging. The development interface is accessed by the debugging software via a JTAG port. Through the development interface, the debugging software can analyze the status of the CPU, memory contents, trace information, etc. The debug and development interface description takes about 2600 SystemC code lines.

4.4. Other units

The exception management unit of this CPU can raise ten different types of exceptions that can be hardware-caused or software-caused. This unit is described in about 600 SystemC code lines.

The OpenRISC 1000 architecture only provides one line of external interrupt. Thus a programmable interrupt controller (PIC) has to be included when more interrupt lines are needed. Other interrupt sources are the tick timer (software oriented clock) and watchdog. These modules have been described in 466 SystemC lines.

The performance counter unit keeps a count of the

number of times that a certain event has occurred. These events are: instruction fetches, load and store accesses, cache misses and watch points. The programmer can obtain profiling information about the software executed by checking these counters. The performance counter unit has been described in 734 SystemC code lines.

The power management unit can modify the system clock frequency, shut down modules or force the CPU to enter sleep mode in order to reduce the power consumption. The software can access all these features. The module description has only 138 code lines.

5. Software Development Kit

Our first idea was to re-use the set of software development tools (assembler, linker, C/C++ compiler, debugger and architectural simulator) that the OpenRISC distribution includes and focus our work on the eCos RTOS porting. However, those tools are neither very stable nor well-tested, and with daily use they prove to be prone to errors that are not acceptable for the development of an industrial application. In fact, even the Application Binary Interface (ABI) was not totally defined and some work with the OpenRISC Team was needed to fix it. In order to find a solution for these problems, a profound analysis of each of the tools has been made. The C/C++ compiler (GCC 2.95) seemed to have some errors because the OpenRISC port uses pieces of code borrowed from ports of other architectures, in an effort to obtain a working compiler in the shortest possible time. Additionally, many developers have been simultaneously working on the code without clear guidelines. The chosen solution was to rewrite the port, emphasizing on producing a clear and robust code. In order to develop a new port, both a set of target description macros (which summarize basic characteristics of target architecture) and a machine description file (that defines the way to translate from the parsed C/C++ code to OR1K assembly) need to be written (about 2000 code lines and 4 man-month effort). This port is being integrated with the latest version of the OpenRISC port and included in the OpenRISC distribution.

The available debugger (GDB) has a very limited functionality and it fails when complex tasks, such as stack backtrace or function identification, are performed. The reasons for these problems are an incomplete development status, stressed by a strong dependency on the previously commented unstable compiler. Therefore, a complete rewriting was carried out, synchronizing the debugger with the new compiler and integrating it within the new GDB multi-architecture framework[16]. Additionally, a software module to communicate the GDB with the in-circuit development interface has been developed.

Existing binary utilities for OpenRISC (mainly assembler and linker) are in working status. Thus, the distribution version works correctly with the new compiler and debugger, and it is being used in our design flow.

The simulator is a key element for both embedded application developers and platform designers. The former use it to verify the functionality of the application software, while the latter utilize it as a golden model of the platform. The simulator provided by the OpenRISC Team (or1ksim), is a classical Instruction Set Simulator (ISS) that lacks the ability to measure performance improvements gained by means of architectural modifications or to provide cycle-accurate information. In order to fix these problems, a cycle-accurate simulator has been developed (more than 10000 C code lines, including debugger support).

The Embedded Configurable Operating System (eCos) [19] was chosen as embedded RTOS, mainly due to its small memory size which makes it an ideal choice for low-cost embedded systems. Additionally, the OpenRISC core configuration has been integrated within the eCos Configuration Tool environment, merging both tasks and avoiding misconfigurations between hardware and software. In order to port eCos to the OR1500 based platform, a new Hardware Abstraction Layer (HAL)[19] had to be developed (more than 2500 C and assembler code lines). This RTOS also include an important set of test benches that have allowed the verification of the port, the SDK and the complete hardware platform.

6. Verification Methodology

One of the most important aspects to guarantee the quality of an IP core is the verification methodology. As was previously commented, the OpenRISC verification methodology was too poor to validate the new core, thus a new verification environment had to be developed. Our verification methodology defines 3 verification levels.

6.1. Block level verification

Every system component must have its own verification environment. Classical signal-oriented tests or transaction-based tests are used to verify the system blocks. The modeling style defined in the SystemC Verification Guide [20] has been used to generate the transaction-based tests. The use of transactors meant an important reduction of the test environment size, test documentation and modification, with the consequent reduction of the verification effort. Coverage metrics have

been used to certify the test quality. In this project, the block test environments achieve 100% line coverage in all the blocks. The GNU 'gcov' tool has been used to calculate this coverage. Table 1 shows the total number of block-level test bench lines, classified by module.

Table 1: Block and module oriented tests

Modules	Number of Blocks	Block Test Bench lines	Module Test Bench lines
Integer unit	11	2007	1056
Excep. unit	1	284	284
Data cache	3	171	903
Ins Cache	2	198	747
Debug unit	1	1930	1930
Dev. Interface	2	72	2044
Perf. Counters	1	359	359
Power Manag.	1	190	190
Tick Timer	1	135	135
Watchdog	1	99	99
PIC	1	120	120

6.2. Module level verification

At module level, functional test benches and random tests have been used. These tests verify the block relationships and detect special corner cases. It is interesting to highlight that the transaction-based random tests [18] have detected more than 30 very specific corner cases with a simple test infrastructure. The SystemC Verification Library (SCV [20]) has allowed us to define very complex random tests (e.g. correct instruction sequences) very easily and with a low effort. At module level, the behavior of the RT-description is compared with the cycle-accuracy architectural simulator, providing an automatic checking of the RT core outputs. Thus, the simulator has been used as a golden model of the system. To increase the simulator confidence, different engineering teams have developed the simulator and the RT-model. Table 1 shows the total number of lines of the module-oriented verification environment.

6.3. System level verification

At this level, some test programs are used to verify the platform. Our verification environment includes two types of test programs: functional tests and application example programs. About 220 assembler functional test programs have been developed to validate particular aspects of the platform. Engineers, who were not involved in the CPU design, derived these tests from the system specification document. Additionally, a new test is included every time that a new bug is detected (regression test). The main application-example test is the complete set of eCos tests. It includes 154 tests that explore all the RTOS possibilities. These tests have also been used to verify the GDB interface with the development interface through JTAG core facilities.

7. Results

In this section several aspects of the developed platform will be analyzed. Firstly, the architectural design of the microprocessor is evaluated. The new microprocessor core (OR1500) needs on average 17% less clock cycles than OR1200 when it is configured without caches. With caches, the performance gain is close to 40%. Considering that a 3-clock-cycle delay is needed to access the main memory, a 40MHz implementation of OR1500 will reach 34.6 MIPS. Without caches, the microprocessor reaches 6.2 MIPS.

Secondly, the simulation performances of the core is commented. As has been previously commented, two models have been developed: a functional model (that includes the architectural simulator, ISS, and functional models of other platform components) and a synthesizable RT-level model. The functional mode reaches about 850000 instructions per second while the RT-level model reaches about 7400 instructions per second. Thus, the functional model is about 115 times faster than the RT-level model although it maintains the cycle accuracy. These simulation results have been obtained in a 2GHz PC, with Linux OS. The SystemC RT-level simulation time is close to the Verilog simulation time (using the Synopsys VCS simulator) although the Verilog simulation could sometimes be up to 10% faster. The Verilog descriptions have been automatically generated from the SystemC code using the Synopsys SystemC Compiler.

After the simulation performance analysis, the synthesis results of the platform are presented. The system has been described in about 14500 synthesizable SystemC code lines. This code was automatically translated to Verilog (as was previously commented) and synthesized with Synopsys tools. The target technology was a Xilinx Virtex2 FPGA with a medium speed grade (-5). Table 2 shows the area and critical paths of four configurations: Minimal (only the integer unit without multiplier), Minimal with multiplier (hardware resources for one-cycle multiplication and MAC instructions are included), common (8Kbyte data and instruction caches are included) and maximum (all the optional modules, such as debug unit, development interface, performance counters, etc, with maximum configuration parameters are included). Now, the platform is being implemented in a FPGA development board and verified in a real demonstrator.

Table 2. Synthesis results

Configuration	Area (gates)	Critical path
Minimal	67481	17.5 ns
Minimal with multiplier	80619	19.2 ns
Common	1166541	24.6 ns
Maximum	1241016	24.6 ns

Finally, some comments about the design effort. We have estimated that about 4.5 man-years have been spent in platform development. About 30% of the effort has been spent in the SDK and the rest (70%) in the hardware development. The software effort has been distributed between the simulator (30%), eCos port (28%), gcc port (22%) and gdb port (20%). The GDB port includes the communication with the in-circuit development interface. The hardware effort has been distributed between the platform design (48%), verification (42%) and synthesis (10%). We also estimated that the use of an open-source core has reduced the total development effort by 50% even although a huge modification has been performed.

8. Conclusions

A first conclusion of this work is that an open-source core (such as OpenRISC) has enough quality to be integrated in an industrial project and it is a very good way to develop modified cores that cover specific company requirements at low cost. Additionally, we think that an open-source core is not really reusable until at least two different implementations have been performed. Most of our problems have been a consequence of the fact that the original OpenRISC SDK source code has not been reviewed deeply enough and the core documentation has not been properly updated. After the profound revision and code improvements that this work has introduced in the open-source core, the distribution (especially the SDK) can be reused and modified with less effort.

We have developed a new platform that covers most of the predefined requirements (low cost and high performance). This platform is based on a new microprocessor core, OR1500. This core, and in general any microprocessor IP, is not a "normal" core because it needs a complex software development kit to support it (compiler, debugger, RTOS, simulators, performance analysis tools, etc). Thus, this type of cores needs particular quality metrics and design reuse methodologies. The platform has been described in synthesizable SystemC code and implemented in an FPGA. The use of SystemC has had a very positive impact in the project, especially in the verification part. The integration of C/C++ programs (such as the architectural simulator) into the SystemC verification environment has been very easy, direct and efficient and the SystemC Verification Library (SCV) has allowed the reduction and clarification of all the test benches with an important decrease in the verification effort.

9. References

[1] *The International Technology Roadmap For Semiconductors*. 2001 Edition. http://public.itrs.net/Files/2001ITRS/home.htm

[2] http://www.vsi.org/

[3] P. Bricaud, M. Keating. *Reuse Methodology Manual*. Kluwer Academic Publisher. 1998.

[4] http://www.us.design-reuse.com/

[5] G. Saucier, T. Pfirst, M. Have, M. Radetzki, P. Neuman. *IP Transfer: a mapping problem*. IP-Based SoC Design Workshop. 2002.

[6] W. Savage, J. Chilton, R. Camposano. *IP Reuse in the System on Chip Era*. ISSS'00. 2000.

[7] D. Lammers. *VSIA's new leader has 'revitalization' plan*. Eedesign. March 17, 2003.

[8] A. Dey, J. Moudy. *Cost Savings via Reuse*. Electronic Design Process Workshop (EDP), 2002

[9] G. Moretti. *Your Core – My Problem? Integration and Verification of IP*. Panel of the 38th Design Automation Conference. DAC'01. 2001.

[10] J. Ganssle. The Failure of Reuse. Embedded.com. December 14, 2001.

[11] *OPENCORES – Project: OpenRISC 1000.* http://www.opencores.org/projects/or1k/

[12] *RTEMS.* http://www.rtems.com/RTEMS/rtems.html

[13] *uClinux – Embedded Linux Microcontroller Project – Home Page.* http://www.uclinux.org

[14] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores, September 7, 2002

[15] J.L. Hennessy and D.A. Patterson. Computer Architecture. *A Quantitive Approach*. Morgan Kaufmann Publishers, San Mateo, CA, second edition, 1996

[16] Andrew Cagney. *What is multi-arch.* Cygnus Solutions, 1999. Available:http://sources.redhat.com/gdb/papers/multi-arch/whatis.html

[17] T. Anderson. A preview of VSIA Functional Verification Deliverables. VSIA European Forum. March 3, 2003.

[18] J. Rose, S. Swan. SCV Randomization. August 13, 2003. http://www.testbuilder.net/reports/scv_randomization.pdf

[19] Anthony J. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.

[20] *SystemC Verification Standard Specification*. December 8, 2002. www.systemc.org.

[21] *http://www.openmore.com/*

[22] OpenCores Coding Guidelines. Rev 1.2. July 14, 2003. http://www.opencores.org/tmp/cvsget_cache/common/opencores_coding_guidelines.pdf

[23] R. Usselmann. Verification Strategies. Rev 0.1. February 4, 2001. http://www.opencores.org/tmp/cvsget_cache/common/ver_plan.pdf