Verifying Open Source CPU Cores using Instruction Set Simulators in OVM Environments

Waqas Ahmed[1], Siegfried Brandstätter[2] and Mario Huemer[3]
[1]Royal Institute of Technology (KTH), Stockholm, Sweden
[2] DMCE GmbH & Co KG, Linz, Austria
[3]Klagenfurt University, Klagenfurt, Austria
WaqasW@kth.se, SiegfriedX.Brandstaetter@intel.com and Mario.Huemer@ieee.org

Abstract

This paper deals with a verification methodology and environment for open source central processing units (CPUs). The aim is to save development time and verification effort by (i) implementing a flexible and reusable verification environment by means of Open Verification Methodology (OVM) and (ii) using the instruction set simulator (ISS) of the CPU core as a golden model. Applying this methodology implements a unified verification environment which facilitates a comparative verification of the CPU core and its ISS on a single platform. Hence, it significantly saves resources to implement the golden model and to perform the verification of the ISS. Moreover, reusing the ISS as a golden model considerably speeds up the CPU design process. We developed a platform to verify the open source OpenRISC1200 (OR1200) core and its ISS as a benchmark.

1 Introduction

According to [14], the performance of modern processors doubles every 18 months by exploiting several mechanisms like out-of-order execution, on-chip caching, speculative execution, prefetching and thread switching. Of course, these techniques increase the complexity of the processors [10] as well as the complexity of verification. Thus, the verification of processors requires a great engineering effort and time. It consumes 50% to 70% of the design resources (time and effort) and is considered to be a bottleneck in the development of modern computing systems [15]. Hence, a more innovative and practical approach is needed to verify complex designs and to keep expenses within the budget. The basic questions which arise for the verification of a hardware design are: (i) what is the most effective and appropriate verification method, (ii) how can the reusability of the verification effort be increased and (iii) how can the verification cost (time and effort) be reduced.

It has been proven that traditional verification methods of writing directed tests in assembly or in high-level languages (C/C++) are insufficient for the verification of complex designs [16]. The reason is that the results of directed tests need to be known in advance for comparison. Hence, this verification approach is not suitable for CPU cores because of their complex instruction sets [6][16]. Another choice is to perform formal verification and mathematically prove a given system [7]. Although this approach offers high verification coverage, it is very complex to apply for designs of large size [3]. However, the functional verification is an essential verification methodology for complex designs [6]. This methodology increases the productivity of a SoC design on a higher abstraction [17]. A coverage-driven constraint random test generation is not only an appropriate functional verification approach for complex designs but also offers high verification coverage. Therefore, we opted this approach for the functional verification of the OR1200 core.

The lack of flexibility and reusability in the development of verification environments consumes more resources and keeps the verification costs high [15]. Hence, reusing the verification blocks will significantly reduce the development time and effort. It would be of great value if flexible verification environments are developed which employ reconfigurable and reusable verification components.

In this paper, we present a flexible verification environment which is created by means of OVM. This allows the users to develop modular and reusable verification components and environments by providing a methodology and a SystemVerilog based supplementary class library [5][11]. Since all components in OVM based verification environments interact with each other via standard transaction-level modeling (TLM) interfaces, such environments are very easy to build and maintain.

Although the verification is performed at every abstraction level of a design, the architectural verification of register transfer level (RTL) is particularly difficult because of the unavailability of a high-level description for the comparison of the results [4]. There are mainly two approaches for this purpose. One is to implement a golden model of the design for the comparison of the simulation results. The other one is to write assertions for the entire design. The latter approach is rather inconvenient because well-defined specifications of designs at RTL abstraction are mostly not available. Hence, the implementation of a golden model is mostly needed. As a matter of fact, the implementation of a CPU core's golden model is a very complex task.

In this paper, we propose a verification methodology in which the ISS of a CPU core is used as a golden model. To follow this methodology, a CPU core and its ISS need

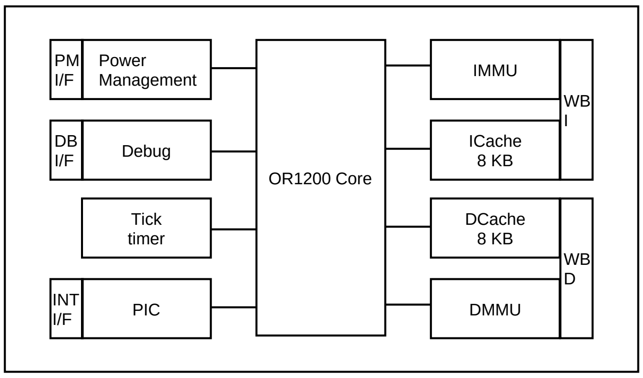

Figure 1. Block diagram of the OR1200 processor.

to be available. Typically, the ISS is implemented before the core for performance analysis on the instruction set. Thereby, the proposed methodology of reusing the ISS as a golden model not only speeds up the implementation and verification process of a CPU core but also saves resources for implementing a golden model.

In this paper, an OVM based verification environment is developed by following the methodology described above. It performs a coverage-driven constraint random functional verification of the OR1200 core (RTL model) where the ISS (Or1ksim 0.3.0) of the core is used as a golden model.

2 OR1200 Core

2.1 Overview

We performed the functional verification of the OR1200 core which is the CPU of the OR1200 processor as shown in Figure 1. The OR1200 processor is an open source soft-processor under the LGPL license. Along with the core, the processor also provides additional utilities including a debug unit, a high-resolution hardware timer, an interrupt controller and a power management unit. The OR1200 core is a 32-bit scalar RISC core with a Harvard memory architecture. It has a single-issue 5-stage integer pipeline, virtual memory support and a multiply-accumulate (MAC) unit for basic digital signal processing (DSP) operations. The OR1200 core delivers a sustained throughput and supports single-cycle execution for most of its instructions. The core is connected to external peripherals and memories through two Wishbone interfaces, the data interface (DWB) and the instruction interface (IWB).

2.2 Instruction Pipeline Architecture

The OR1200 core implements a 5-stage integer pipeline as shown in Figure 2. The Instruction Fetch (IF) stage is the first pipeline stage followed by the Instruction Decode (ID), the Execute (EX), the Load/Store (LS) and the Write-Back (WB) stage. The instructions are fetched from the memory system and dispatched to the corresponding

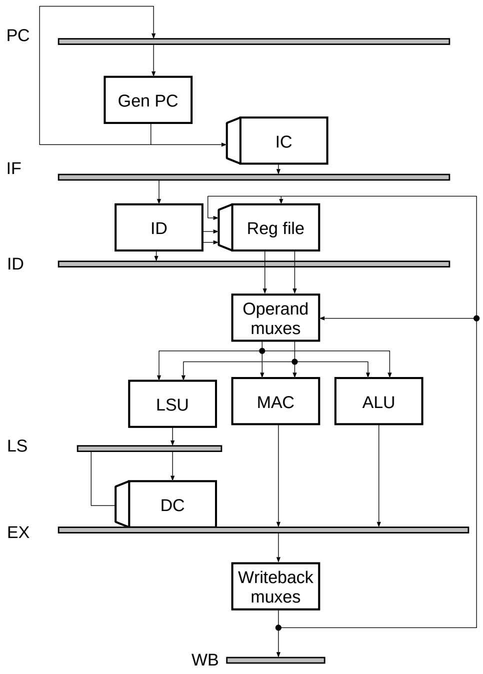

Figure 2. Register abstraction of the OR1200 pipeline [1].

execution units: (i) the load/store unit (LSU), (ii) the arithmetic logic unit (ALU) and (iii) the MAC unit. A precise exception-model is implemented in parallel to control the pipeline.

The functional verification of such a deeply pipelined CPU core is a challenging task. It is considered to be the most complex and expensive task in the development of modern SoC designs [13]. The verification of the OR1200 core is even more complex because of its complex instruction set. There are five instruction formats and two addressing modes. The instruction set mainly consists of single- and multi-cycle instructions, jumps and branches followed by a delay-slot, and MAC instructions. Accounting the dependencies between these instructions in the pipeline and the exception handling, the verification becomes even more complex. Furthermore, the simulation time to run test sequences and to get a satisfactory verification coverage is a matter of high significance, particularly while verifying pipelined cores like the OR1200 which has a large number of registers and a complex instruction set.

3 Verification Environment

3.1 Overview

We used OVM to implement a reconfigurable and reusable verification environment for the simulation based verification of the OR1200 core. It implements a constrained random generation of verification scenarios and a vibrant coverage model including a scoreboard to assess the verification completeness. Figure 3 elaborates the

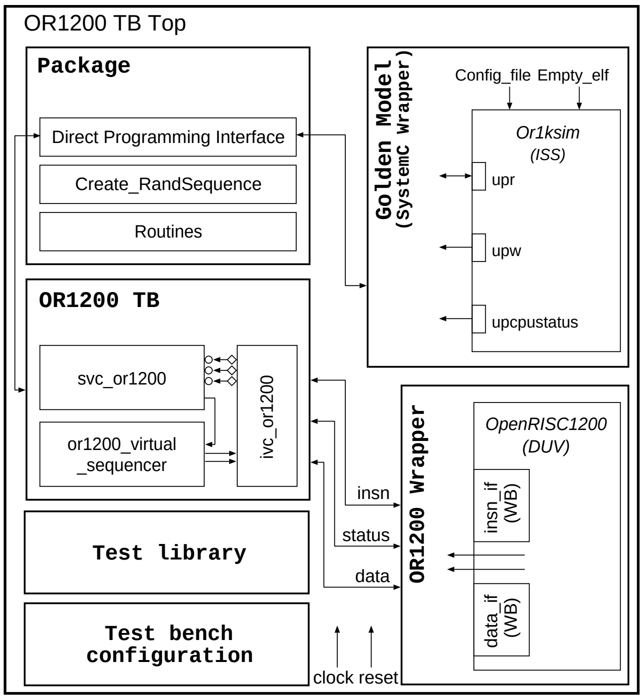

Figure 3. Verification environment for the OR1200 core.

architecture of the verification environment (*OR1200 TB Top*) which includes

- the *Golden Model*,
- the device under verification (DUV) wrapper (*OR1200 Wrapper*),
- the main test bench (*OR1200 TB*),
- the global package (*Package*),
- the *Test library* and
- the *Test bench configuration*.

3.2 Golden Model

The ISS of the OR1200 core, which is used as a golden model, can be compiled as an executable or as a library [2]. In the verification environment the ISS is used as a library. Since it is not intended to execute an application program, an empty ELF image is provided to the ISS. By default the ISS library provides two upcalls (*upr*, *upw*) to its public interface to read/write the external peripherals [8]. These upcalls are modified to feed the ISS with instructions and data for load instruction, and to read data for store instructions. A third upcall (*upcpustatus*) is implemented to write the ISS status up to SystemC wrapper after the execution of each instruction. The communication between the ISS and the test bench is synchronized by means of SystemC FIFOs. The main test bench, implemented in SystemVerilog (OVM), accesses these FIFOs through Direct Programming Interface (DPI) functions. More details about the DPI can be found in [9].

3.3 DUV Wrapper

A SystemVerilog based wrapper (*OR1200 Wrapper*) is implemented around the OR1200 core for a structural connectivity as shown in Figure 3. This wrapper provides interfaces for instructions, data and the status of the OR1200 core. These interfaces are used to access the IWB interface, the DWB interface and the internal signals of the core, respectively. The status interface of this wrapper makes all required internal signals of the DUV available at its ports. The internal signals include the status registers (to be monitored) and the control signals (to control the monitoring). The status of the OR1200 core is read over simulator functions. The wrapper manipulates the internal control signals according to the requirements e.g., delay a control signal for two clock cycles. The wrapper also performs the translation of the internal signals to usable formats. All components of the verification environment interact with the DUV only through the wrapper's interfaces.

3.4 Main Test Bench

The main test bench (*OR1200 TB*) is a reconfigurable and reusable component which is developed by means of OVM. It interacts with the golden model through its *imported DPI functions* and uses its physical interfaces to interact with the DUV wrapper. The main test bench executes configurable tests generated by the *test library*. All tests are a constrained random generation of scenarios comprised of OR1200 instructions. Every verification component at any hierarchical level is reusable and can be configured by the *test bench configuration*. For example, (i) whether the coverage model or the scoreboard is implemented or not, (ii) whether an agent component operates as a passive component, or (iii) which tests of the test library are executed. All components interact with each other through standard TLM interfaces. The main test bench implements a layered architecture and is comprised of three main components:

- the interface verification component (*ivc_or1200*),
- the system verification component (*svc_or1200*) and
- the virtual sequencer (*or1200_virtual_sequencer*).

The main test bench first sends an instruction to the golden model, writes/reads data (if the instruction is a Load or a Store instruction) and receives the ISS status once the instruction has been executed. This instruction is then sent to the DUV. Since the DUV is a pipelined implementation, the main test bench implements a synchronization mechanism by examining the control state machine of the DUV along with the data-path and reacting accordingly. It copes with the core's pipeline problems (delays) by monitoring the control signals of the DUV. It determines the exact time to examine the status of the DUV, which is indicated by the program counter (PC) and the special purpose registers (SPRs), and the execution results

Table 1. Malfunctions in the OR1200 core and the ISS.

Instructions	**Error Description**
Extend Byte with Sign (l.extbs) Extend Byte with Zero (l.extbz) Extend Half Word with Sign (l.exths) Extend Half Word with Zero (l.exthz)	All these instructions are working correctly in the ISS but are not implemented in the OR1200 core. If any of these instructions is sent to the core, another instruction "l.movhi" is executed instead of generating an illegal instruction exception. Hence, the execution of an unimplemented instruction is not reported and an incorrect result is calculated.
Add Signed and Carry (l.addc)	Since the carry flag is not controlled by the freeze logic in the OR1200 core, a wrong value is added to the result. Moreover, the carry flag implementation in the ISS is also erroneous.
Divide Signed (l.div) Divide Unsigned (l.divu)	According to the specification of both divide instructions the carry flag should be set if the divisor is zero. However, the ISS generates an illegal exception if the divisor is zero. The OR1200 core neither generates an illegal exception nor sets the carry flag. This is a clear discrepancy between the specification of the divide instructions and their implementation in the OR1200 core and the ISS.
Find Last 1 (l.fl1)	The instruction is neither implemented in the OR1200 core nor in the ISS. However, when this instruction is sent to the core, another instruction "Find First 1 (l.ff1)" is executed instead of generating an illegal instruction exception. Hence, the execution of an unimplemented instruction is not reported and an incorrect result is calculated.
Multiply Immediate Signed and Accumulate (l.maci)	The instruction is decoded correctly neither in the OR1200 core nor in the ISS. Therefore, a wrong immediate value is used in the calculation.
Multiply Immediate Signed (l.muli)	The instruction is not working correctly in the OR1200 core. It is a multi-cycle instruction but not controlled by the freeze logic. Therefore, an incorrect result is selected.
Multiply Unsigned (l.mulu)	The instruction is neither implemented in the ISS nor in the OR1200 core. According to the specification, this instruction is compulsory to implement [12]. Despite that, when this instruction is sent to the core, a wrong implementation is executed instead of generating an illegal instruction exception. Hence, the execution of an unimplemented instruction is not reported and an incorrect result is calculated.
Jump Register and Link (l.jalr) Jump Register (l.jr)	The effective address for both jump instructions is the content of a GPR which can be an unaligned address. The instruction fetch in the OR1200 core is naturally word-aligned but not in the ISS. Moreover, the ISS does not implement exception handling in case of an unaligned access to fetch a new instruction.
Add Immediate Signed and Carry (l.addic)	The instruction is implemented in the OR1200 core but not in the ISS. Although the instruction generates correct results in the OR1200 core for directed tests, its correctness is not proven since it could not be included in the exhaustive verification test.
MAC Read and Clear (l.macrc)	The instruction is not working correctly in the ISS. Although the instruction generates correct results in the OR1200 core for directed tests, its correctness is not proven since it could not be included in the exhaustive verification test.
Rotate Right (l.ror) Rotate Right with Immediate (l.rori)	Both instructions are implemented in the OR1200 core but not in the ISS. Although both instructions generate correct results in the OR1200 core for directed tests, their correctness is not proven since they could not be included in the main verification test.
Move to Special Purpose Registers (l.mtspr) Move from Special Purpose Registers (l.mfspr)	The ISS implementation of both instructions defines a wrong address for accessing the special purpose register. The implementation in the OR1200 core is correct but not proven because it could not be included in the main verification test.
Unimplemented Overflow Flag (OV)	According to the OR1200 architectural manual a number of instructions can drive the OV flag. However, the OV flag is not implemented in the OR1200 core.

within the general purpose registers (GPRs). Then it compares the status of the golden model with the DUV status and scoreboards it. The main test bench also implements a coverage model to assess the completeness of the verification.

In case of reusing this verification environment for verifying further open source CPU core, the main effort will focus on the timing synchronization between the core and its ISS. The rest of the components can be easily reconfigured and employed.

4 Verification Results

The verification results of the OR1200 core show that the core has some malfunctions including (i) erroneous instructions, (ii) unimplemented instructions, (iii) design errors and (iv) discrepancies between the specification and its implementation. Moreover, the OR1200 ISS has some implementation errors and unimplemented instructions, too. Since these instructions have to be excluded from the verification, it significantly restricts the achievable veri-

Table 2. Coverage results of the OR1200 core.

OR1200 instruction types	Total number of instruction	Executed number of instruction	Achievable coverage (%)	Reached coverage (%)
Insn rD, rA, rB	17	11	64.7	60.2
Insn rA, rB	12	12	100	100
Insn rD, rA, I	13	9	69.2	69.2
Insn rA, I	10	10	100	100
Insn I (rA), rB	4	3	75	75
Insn rD, rA, L	4	3	75	75
Insn N	5	5	100	100
Insn rD, K	1	1	100	100
Insn rD, rA	4	0	0	0
Insn rB	2	0	0	0
Insn rD	1	0	0	0
Insn rB, I	1	0	0	0
Driving carry flag	10	4	40	40
Driving flag	20	20	100	100

fication coverage of the OR1200 core. The verification results are summarized below.

4.1 Malfunctions in the OR1200 core and the ISS

The Table 1 presents a summery of all errors and faults discovered in the OR1200 core and its ISS.

Several benchmark programs were compiled using the OpenRISC32 C/C++ compiler but it did not generate most of these erroneous instructions. This means that the compiler either does not implement these instructions or does not often generate them. This is the reason why the errors within these instructions stayed unidentified before. However, the OR32 assembler is able to assemble these instructions.

4.2 Verification Coverage Results

This section presents the verification coverage of the OR1200 core.

The main scope of this verification coverage is to show the verification completeness with respect to the decoding of instructions within the core. Therefore, the coverage matrix corresponds to the instruction decode and pipeline control logic. Verifying the correctness of the execution of instructions with respect to data values is not included due to the state explosion problem.

No instruction or scenario having a problem either in the ISS or in the OR1200 core is included in this verification since the errors have not been corrected. There are 78 instructions in the OR1200 instruction set but only 58 instructions could be included in the verification test because 20 instructions are erroneous or unimplemented (either in the OR1200 core or in its ISS). Hence, the overall instruction verification coverage is restricted to 74.3% which was successfully achieved.

The Table 2 shows the verification coverage results of the OR1200 core. There are several instruction types in the OR1200 instruction set where each OR1200 instruction belongs to one of these types. The table shows the total number of instructions belonging to a particular instruction type along with the number of instructions that could be executed in the main verification test. The results show the verification coverage achieved for each instruction type which is based on the number of executed instructions. The maximum achievable coverage is reached for all instruction types except for the instruction type *“Insn rD, rA, rB”*. This is because of its large test space which is composed of 17 instructions and three GPRs i.e., $17 \times 32 \times 32 \times 32$ combinations. The maximum achievable coverage for this instruction type is 64.7% while only 60.2% could be reached. There are four instructions namely *l.nop*, *l.csync*, *l.msync* and *l.psync* which are separately verified and working correctly in the core and the ISS.

The cross coverage of three contiguous instructions in pipeline stages of the OR1200 core is also taken into account to observe the dependencies between instructions. The maximum achievable cross coverage is 40.7% while only 40.0% could be reached because of large test space and unreached combinations of the instruction type *“Insn rD, rA, rB”*.

5 Conclusion

The verification of processors consumes upto 70% of resources (time and effort) and is acknowledged as a major bottleneck in the development process, according to a variety of publications. This paper presents an OVM based reusable verification environment for coverage driven constrained random verification of the OR1200 core. Furthermore, a new verification methodology is introduced by using the ISS of the core as a golden model. This methodology is successfully proven because it has credibly verified not only the OR1200 core but also the ISS on a single platform. Since the ISS is often implemented before the core, it can be easily reused as a golden model. This will significantly reduce the development time and the verification effort. Hence, the methodology is also proven beneficial because it saved resources to verify the ISS and to develop the golden model for the core.

6 Acknowledgements

This work was funded by the COMET K2 Center "Austrian Center of Mechatronics (ACCM)". The COMET program is funded by the Austrian government, the federal state of Upper Austria and the scientific partners of ACCM.

References

[1] W. Ahmed. Implementation and verification of a cpu subsystem for multi-mode rf transceivers. Master's thesis, Royal Institute of Technology (KTH), June 2010.

[2] J. Bennett. *Or1ksim User Guide*. Embecosm, 2009.

[3] J. Bhadra, M.S. Abadir, L.-C. Wang, and S. Ray. A survey of hybrid techniques for functional verification. In *Design & Test of Computers, IEEE*, volume 24 , issue:2, pages 112 – 122, June 2007.

[4] M. Bose, J. Shin, E.M. Rudnick, T. Dukes, and M. Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 442 – 448, August 2001.

[5] Cadence. *Open Verification Methodology User Guide*. Cadence Design System, version 2.0 edition, September 2008.

[6] J. C. Chen. Applying constrained-random verification to microprocessors. *EE Times EDA Designline*, October 2007.

[7] D. Deharbe, S. Shankar, and E.M. Clarke. Formal verification of vhdl, the model checker cv. In *Proceedings of the XI Brazilian Symposium on Integrated Circuit Design*, pages 95 – 98, October 1998.

[8] Embecosm. *The Or1ksim Simulator*. Embecosm.

[9] Mentor Graphics. *ModelSim User's Manual*. Mentor Graphics, May 2008.

[10] D. Haakon, G. Marius, and N. Lasse. Cache write-back schemes for embedded destructive-read dram. In *Proceedings of the 9th International Conference on Architecture of Computing Systems (ARCS)*, pages 145–159. 2006.

[11] S. Imam. *Step-by-Step Functional Verification with SystemVerilog and OVM*. Hansen Brown Publishing Company, 2008.

[12] D. Lampart. *OpenRISC 1000 Architecture Manual*. OpenCores, April 5 2006.

[13] P. Mishra, N. Dutt, and Y. Kashai. Functional verification of pipelined processors: A case study. In *Proceedings of the 2005 IEEE/LEOS Workshop on Fibres and Optical Passive Components*, pages 79 – 84, September 2004.

[14] P. P. Ravale and S. S. Apte. Design of a branch prediction unit of a microprocessor based on superscalar architecture using vlsi. In *2nd International conference on Computer Engineering and Technology (ICCET)*, pages V3–355 – V3–360, June 17 2010.

[15] A. Sagahyroon, G. Lakkaraju, and M. Karunaratne. A functional verification environment. In *48th Midwest Symposium on Circuits and Systems*, volume 1, pages 108 – 111, August 2005.

[16] F. Vitullo, S. Saponara, E. Petri, M. Casula, L. Fanucci, G. Maruccia, R. Locatelli, and M. Coppola. A reusable coverage-driven verification environment for network-on-chip communication in embedded system platforms. In *Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 71–77, June 2009.

[17] M.-K. You and G.-Y. Song. Systemverilog-based verification environment using systemc custom hierarchical channel. In *Proceedings of the IEEE 8th International Conference on ASIC*, pages 1310 – 1313, October 2009.