A Synchronization Method for Register Traces of Pipelined Processors

Ralf Dreesen[1], Thorsten Jungeblut[2], Michael Thies[1], Mario Porrmann[2], Uwe Kastens[1], and Ulrich Rückert[2]

[1] University of Paderborn, Department of Computer Science
{rdreesen,mthies,uwe}@upb.de
Fürstenallee 11, 33102 Paderborn, Germany
[2] University of Paderborn, Heinz Nixdorf Institute,
Fürstenallee 11, 33102 Paderborn, Germany
{tj,porrmann,rueckert}@hni.upb.de

Abstract. During a typical development process of an embedded application specific processor (ASIP), the architecture is implemented multiple times on different levels of abstractions. As a result of this redundant specification, certain inconsistencies may show up. For example, the implementation of an instruction in the simulator may differ from the HDL implementation. To detect such inconsistencies, we use register trace comparison. Our key contribution is a generic method for systematic trace synchronization. Therefore, we convert a micro-architectural trace into an architectural trace. This method considers pipeline hazards and non-uniform write latencies. To simplify the validation of a processor, we further have implemented an automatic validation environment that includes a tool which points the developer directly to erroneous instructions. The flow has been validated during the development of our CoreVA architecture for mobile applications.

1 Introduction

Our *processor design flow* is divided into the development of a compiler tool chain and the development of the hardware description (see Figure 1). The compiler tool chain consists of a compiler, an assembler, a linker, various debugging tools and an instruction set simulator (ISS).

All these tools are generated from a central processor specification (UPSLA [7] — Unified Processor Specification Language). This allows the rapid generation of a complete and consistent toolchain, which can be used to perform a *design space exploration* of the processor. Using this approach, we can easily add application specific instructions, re-generate the toolchain and evaluate the processor using the ISS. The consistency within the toolchain is guaranteed by the use of a central processor specification. All aspects (e.g., machine format, write latencies) of an instruction are specified once in this specification and then re-used throughout the assembler, linker, simulator, and disassembler. For the *hardware development* we describe the processor in VHDL and use the standard design flow to obtain an FPGA prototype or the final ASIC implementation of the processor.

A. Rettberg et al. (Eds.): IESS 2009, IFIP AICT 310, pp. 207–217, 2009.

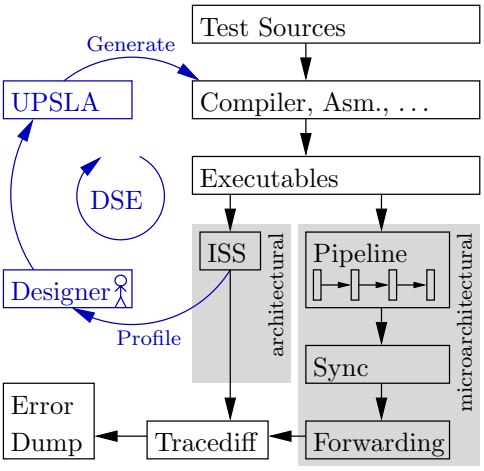

Fig. 1. Overview of our design flow and validation environment

The usability of the toolchain has been proven in multiple industry related projects, in which very long instruction word (VLIW), multiprocessor system-on-chip (MPSoC), or other non-orthogonal processor architectures were developed. The toolchain covers the complete instruction set.

Due to the *separate specification* of the tool chain and the hardware description, inconsistencies may result. Therefore, we need to check the equivalence of the ISS and the synthesized processor. To validate the consistency, we apply a non-formal co-verification approach. Basically, we compare the traces of executing a program on the ISS with the hardware implementation. If the traces diverge, an error in the specification has been detected.

Of course a *non-formal validation by simulation* approach does not guarantee the correctness of the implementation. The effectiveness of the validation heavily depends on the set of *test programs*. The generation of such tests is covered in [5] and is not subject of this paper.

The abstraction level of the ISS implementation differs from the micro-architectural implementation. For example, the micro-architectural implementation executes instructions pipelined, whereas the ISS simulates each instruction as a whole. As a result, the micro-architectural trace differs from the ISS trace and hence *synchronization* is required (Section 3).

As an extension to existing approaches, we do not only perform a *multi-domain validation* of the ISS and the RTL simulation, but also of the processor emulation. Therefore, we have added a trace interface to the processor, which is used to emit the state sequence (Section 4). The ISS is available in an early design stage and essential for a design space exploration. The RTL implementation is more detailed but therefore also more expensive to simulate. The hardware emulation approximates the final ASIC implementation most accurately and is even faster than the ISS and RTL simulation. Some timing issues manifest

themselves at the stage of processor development. The system can also be integrated in real world systems (due to its real time ability). However, the implementation effort for the emulation is high. In addition, the observation of the internal processor state is limited and therefore debugging is cumbersome.

We also attach importance to locating an error, in case that an inconsistency between both traces is detected. Therefore, we have implemented a *specialized comparison tool*, which processes two execution traces and dumps a meaningful description of any detected inconsistency (Section 5).

2 Related Work

A formal verification approach for processor control is presented in [2,8]. Both papers regard a processor model with out-of-order-execution. They conceptually flush the pipeline to synchronize the micro-architectural and the ISA model. We extend this approach by considering pipelining effects caused by forwarding circuits. In addition we also validate the datapath of the processor.

In [3] the trace of an ISS is compared with a Verilog simulator. The Verilog simulator captures the state at the write-back stage at the expense of additional hardware resources. The approach does not consider non-uniform write latencies.

The generation of test cases with coverage feedback is discussed in [5]. For validation a co-simulation approach is used, where the RTL trace is converted into an ISA trace. The authors mention this conversion to be a challenge beyond the scope of their paper. The approach in [10] performs a co-simulation of an ISS and an RTL simulator, both manually written in C. The authors also identified the problem of trace conversion, but do not present a systematic solution. In Section 3 we present a solution to this conversion problem.

The architecture description language "ArchC" to generate an ISS is presented in [1]. To validate the consistency of this simulator and a hardware description language (HDL) implementation, a validation approach which compares memory-transactions is used. We have extended this method by a cycle accurate comparison of registers, including the effects of a forwarding circuit.

The Tensilica tools [4] offer the generation of a toolchain and a hardware description from a single processor specification. Their approach allows the extension of a predefined processor architecture by application specific instructions. However, the core of the processor remains fixed and certain design parameters like instruction format or pipeline depth are not exposed to the developer.

In [9] the authors describe the functional verification of a POWER5 processor. They use a coarse grained memory-trace mechanism that is well suited for system-level verification. The tracing mechanism is tailored to the POWER5 architecture. In contrast, our fine-grained trace-mechanism is generic and focuses on core-level verification.

In summary, trace comparison is a widely accepted approach for processor validation. Prior work has focused on specific architectures and does not offer a *generic method* for synchronization of traces.

3 A Generic Tracing Approach for Pipelined Processors

In this Section we describe a general method for adding a trace interface to a processor. Our method can cope with common processor architectures and pipelining effects like stalls, flushes and non-uniform write latencies. We do also consider forwarding of instruction results. Our approach does not require the trace information to be emitted in a specific pipeline stage. Instead, each signal can be captured at the most convenient stage to avoid additional hardware overhead (Section 3.1). To incorporate the effects of forwarding, we introduce a method to derive the *publication cycle* of an instruction result from its final write-back cycle (Section 3.2). The *publication cycle* is defined as the clock cycle, in which the result becomes visible to other instructions, i.e. when it is fed into the forwarding circuit.

3.1 Synchronization of Traced Signals

A simple implementation for tracing in a pipelined processor would collect all information at the last stage, which is typically the write-back stage. This may introduce additional hardware overhead to pipeline signals that are otherwise not needed in the last stage. Examples include the program counter and the memory write ports. Instead, we capture the signals to be traced immediately in the originating stage.

However, this introduces the problem of *synchronizing* the data, which has been captured at different stages. For example stalls may defer the execution of some instructions in the pipeline or a flush may invalidate an instruction. To synchronize the information we use a *virtual pipeline*, which emulates the hardware pipeline. The virtual pipeline is implemented in software and runs on the computer that monitors the device under test (DUT). Therefore, it does not require additional hardware resources. The virtual pipeline also receives the stall and flush signals from the hardware pipeline, to accurately emulate its behavior.

The example in Figure 2 juxtaposes the operation of the hardware pipeline and the virtual pipeline. In this example, stages 0 and 2 of the hardware pipeline emit trace information which is passed to the respective stage of the virtual pipeline. The trace information A_i represents a set of signals that were emitted in stage i.

In cycle 0 instruction A is executed in stage 0 and emits information A_0. In cycle 1, instruction A is passed to the next stage, just like the information A_0 in the

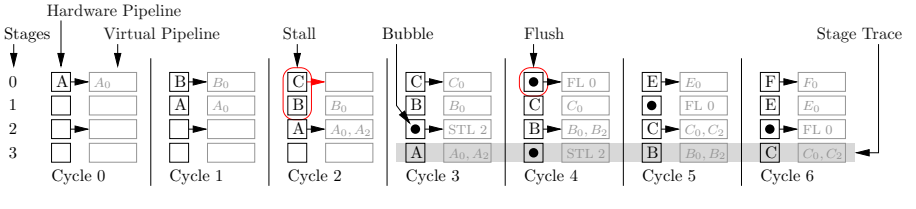

Fig. 2. Example of virtual pipelining

virtual pipeline. In cycle 2 the stages 0 and 1 are stalled. Stage 0 does therefore not emit information and a *bubble* is inserted in stage 2 of the hardware pipeline. The *bubble* floats through the pipeline like a normal instruction without carrying any useful work. The virtual pipeline labels its bubble with the originating stage (STL 2). This allows us to reconstruct the stall and flush events. In cycle 4 stage 0 of the hardware pipeline is flushed and the virtual pipeline records the number of the flushed stage (FL 0). In cycle 5 and 6 execution advances normally.

The final synchronous trace information is collected at the last stage of the virtual pipeline. We will call this trace the *stage trace* in the following. However, this trace does not reflect forwarding effects, which are covered in the next Section.

3.2 Incorporating Forwarding Effects

In this Section we present a method to derive a *storage trace* from a stage trace. The storage trace lists every instruction result, when it is published and therefore visible to other instructions.

This publication cycle of an instruction result can not directly be derived from the stage trace. It is not possible to calculate the publication cycle by simply subtracting a constant from the stage trace cycle. There are two reasons that prevent such an easy computation, namely *non-uniform latencies* and *stalls*.

In a processor architecture with *non-uniform latencies*, results are published in different pipeline stages. Assume that instruction D in Figure 3 publishes its result in stage 2 and E in stage 3. Hence, the result of D appears in cycle 7 and the result of E in cycle 9, whereas D and E show up at cycle 10 and 11 in the stage trace. The cycle difference between the publication cycle and the stage trace cycle is not constant, but depends on the instruction. We therefore need to know the *publication stage* of each instruction. This information can be extracted from our processor specification.

To accurately handle *stalls*, information about pipelined execution is required. Assume that instruction A and B (see Figure 3) publish their results both in stage 2. The difference between publication cycle and stage trace cycle is 3 for

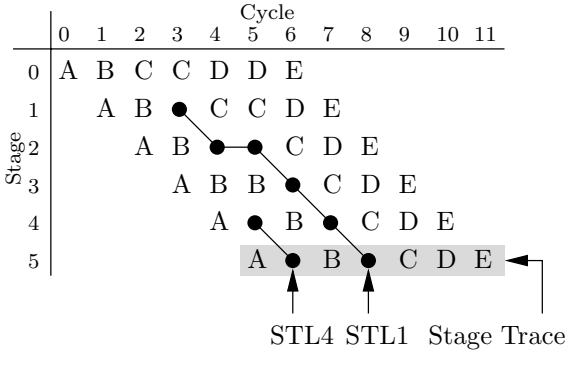

Fig. 3. Pipeline diagram illustrating the instruction execution

instruction A, but 4 for instruction B. If A and B would instead publish their results in stage 4, the difference would be equal. To derive the storage trace from the stage trace, we need to know for every cycle, which instruction is contained in a given stage. Therefore, we need to reconstruct the pipeline execution from the stage trace.

We can *reconstruct the instruction execution* shown in Figure 3 from the stage trace, i.e. the last line of the pipeline diagram. We construct the diagram during a single right to left pass, i.e. from cycle 11 towards cycle 0. The instruction of a given stage s and cycle c can be derived from the next cycle as follows:

$$\text{instr}(c, s) := \begin{cases} \text{instr}(c+1, s) & \text{stalled}(c+1, s) \\ \text{instr}(c+1, s+1) & \text{else} \end{cases}$$

where the predicate stalled is defined as

$$\text{stalled}(c, s) := \exists s' > s : \text{isBubbleOrigin}(c, s')$$

This means that an instruction in a given stage and cycle is equal to the instruction in the next cycle and the next stage, unless there is a stall. In the example in Figure 3 the instruction in cycle 7 at stage 2 is the same as the instruction in cycle 8 at stage 3, as there is no bubble in cycle 8 at stage 4 or 5. The instruction in cycle 4 at stage 0 however is taken from cycle 5 at stage 0, as there is a bubble originating in cycle 5 at stage 4.

The pipeline execution reconstruction gives us the information in which cycle an instruction was executed in a given stage. Together with the information in which *stage* an instruction publishes its result, the pipeline reconstruction allows us to compute the cycle in which the result is published.

4 Emitting Traces

Our multi-domain consistency check is based on the comparison of states. We define the *processor state* as the content of all memory elements in the architecture. These are typically the register files and the data memories. The current state of the processor can be derived from its initial state by continuously tracing all write accesses to its memories. For VLIW architectures we additionally trace the number of the functional unit (FU) which causes a write access. For conditional execution, condition register files can be traced. Also single instruction multiple data (SIMD) mode is considered.

To get a deterministic trace, the *memories must be initialized* at the beginning of the simulation. Even correct programs may load uninitialized data from memory into registers. One example is the copying of partially uninitialized unions in C. Another example is an ISA where a single byte can not be loaded directly, but only by a sequence of a `load word` and `extract byte` instruction.

4.1 Determining the Processor State

To emit a common trace format, we use a single trace library for all three domains. The processor state is passed to a central `trace_cycle` function of this library.

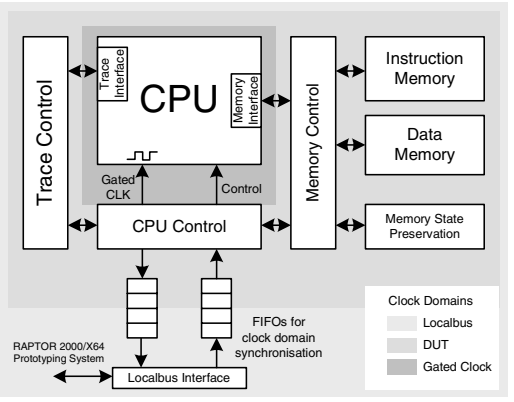

Fig. 4. CPU Control Unit (CCU)

For the ISS the current processor trace can be directly constructed from the memories. To compensate the pipelined execution of instructions in the hardware implementation, we apply our synchronization method as described in Section 3. Results that are written back to the register file are captured in the write-back stage, which is usually the last stage. Memory signals and the program counter are captured at earlier stages, thus requiring synchronization by virtual pipelining.

Branches to unaligned instruction groups (VLIW architecture), accesses to external memory (e.g., cache misses), as well as data or control hazards, can stall pipeline stages of the processor core. These stall signals are forwarded to the virtual pipeline to enable an accurate pipeline emulation.

In the *RTL simulation* we can access the internal signals of the design using the ModelSim Foreign Language Interface (FLI). Just the signal names have to be known by the trace mechanism. The processor core itself is not altered.

To trace processor states from the *hardware emulation*, we offer the designer a generic wrapper (CPU Control Unit, CCU, see Figure 4) to embed the processor core. Only slight changes have to be made to the processor core, by making the signals aforementioned available to the top entity. All signals are connected to register inputs, no additional combinatorial logic is added. As we retrieve most of the signals directly from pipeline registers, tracing has a negligible impact on the timing.

The CCU is mapped to an FPGA and emulated in our rapid prototyping environment RAPTOR2000/X64 (see Figure 5, [6]) This modular system allows the use of a large selection of FPGA daughter boards or physical interfaces (e.g., Ethernet), which enable the use in real environments. Also, the CCU is suitable for ASIC realizations to assist testing. As our prototyping environment features a modular design, the FPGA module can easily be replaced by the ASIC realization after verification. Identical control and test software is used, thus reducing developing time.

Fig. 5. RAPTOR2000/X64 rapid prototyping environment

To gather all information of a processor state, multiple accesses have to be made to the CCU. This requires clock gating for a cycle-by-cycle execution of the processor core. As the latency of synchronous SRAM or caches is usually at least one clock cycle, some additional considerations have to be made. If the processor core performs a read access to instruction or data memory, it applies address and control signals to the memory. For example, if the output of the memory is registered, the requested data is valid one clock cycle later. As we perform a cycle-by-cycle-based execution, this data cannot be stored in the processor pipeline registers in the following clock cycle, so it has to be preserved in the control unit. Between two steps of the processor core, unrelated memory accesses can occur, disturbing the CPU state. Hence, before the next step the state of the memory has to be restored by applying the preserved memory address before the next CPU step.

4.2 Output of Traces

There are two possible ways of passing trace output to the consistency check: by a trace file or online. A trace file enables offline regression tests between the current and a former revision of each domain but has the disadvantage of very large files to be stored and transfered. Even if considering a standard single core as a lower bound, at least the program counter, one register or memory write access (data and address) and some control signals have to be traced, which adds up to about 10 Bytes per clock cycle. For just one simulated second of a 300 MHz CPU, 3 GB of data are generated. Considering more complex architectures, like VLIW or SIMD, multiplies this data. Compression could reduce the size by an order of magnitude but would not completely eliminate the problem of large data size. Another solution is an online consistency check, as used in our approach.

5 Validation Tools

To automate the execution of a large set of test cases, we implemented a *validation environment* as outlined in Figure 1. The environment allows the distributed simulation on multiple systems.

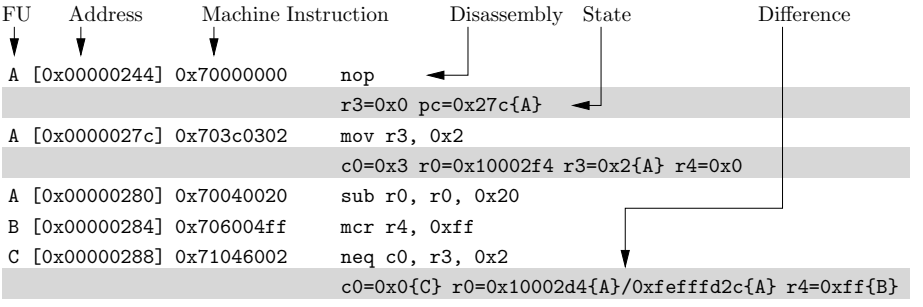

Fig. 6. Example of a tracediff dump

The register traces from the simulators are compared state-by-state using our `tracediff` tool. If states differ, the tool aborts with a meaningful error description which points the developer directly to the location of the error.

The description is a *backtrace* of the last n processor states interleaved with a disassembly of the respective executed instructions as shown in Figure 6. Only those registers that are relevant for debugging are listed. Accesses to the main memory are treated the same way. If the register was written, the respective functional unit is appended in curly braces. If the register value differs in the traces, both values are printed. The disassembly between two states lists one or more instructions that were executed in parallel. Each line is prefixed with the identifier of the functional unit `(A,B,...)`.

In Figure 6 the value of register `r0` which was written by functional unit `A` differs. Considering the `sub` instruction and its input value `0x10002f4`, the result in the right-hand trace is obviously wrong.

6 Evaluation

We have evaluated our system with our CoreVA architecture developed in our research groups. This VLIW processor is a six stage pipelined harvard architecture with non-uniform latencies (see Figure 7). Common features like branch prediction, conditional execution, pipeline forwarding have been implemented. Two load store units access a dual ported memory. 30 general purpose registers (32 bit) can be accessed by ten read and six write ports. Two condition register files (8 bit each) can be accessed by four write ports each. The processor core has been integrated in the CPU Control Unit (CCU). The resource consumption, including 32 kBytes instruction and data memory each, is described in Table 1. The CCU occupies about 6 % of slices of the total system. The FPGA on-chip memory is used as instruction and data memory. The critical path of the processor core is not affected by the connection to the CCU and the trace unit.

For Co-Simulation we use the following setup: On one machine a graphical user interface controls the processor core, and reads out the processor states. On another machine the ISS runs in a Linux environment. Both, emulation

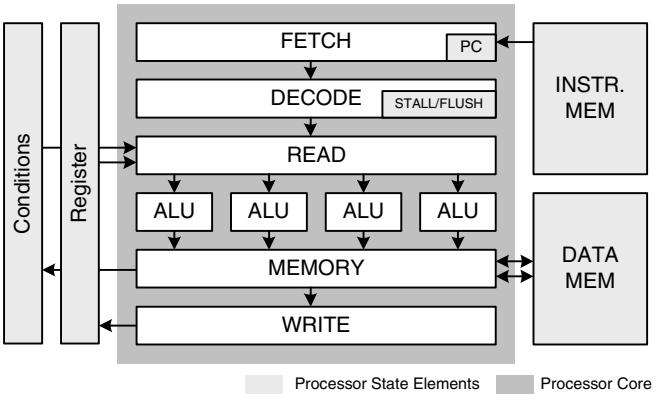

Fig. 7. VLIW architecture

Table 1. Resource consumption of the design mapped on a Xilinx Virtex-II 6000 FPGA

Instance	Registers	LUTS	Slices
RAPTOR Interface	142	36	268
CCU	460	1408	1340
VLIW Core	4875	43063	22344
Total	5477	44507	23952

and simulation, transfer their traces to a third machine, where the validation environment is executed as described in Section 5. Our test repository ranges from microbenchmarks and synthetic benchmarks (e.g., EEMBC, Dhrystone) to real world applications (e.g., 802.11b). Using this setup, we have successfully tracked down an inconsistency due to an ambiguous specification. The error was located in the decoder of the processor's RTL description.

Certainly, tracing of every processor state has an impact on simulation and emulation speed. The ISS and the RTL simulation have to invoke the `trace_cycle()` function. Simulation speed of the RTL implementation reduces by 6% (1.92 kHz vs. 1.83 kHz). For the ISS the execution speed is reduced from 4 MHz to 2 MHz. The hardware emulation has to run the CPU cycle-by-cycle and read out the processor state in between.

7 Conclusion

We have presented a generic validation method for processors to perform consistency checks across multiple simulation and emulation domains. We have developed a generic method for systematic tracing of pipelined processors. The method minimizes hardware overhead, by applying virtual pipelining for the synchronization of traced signals. Processor states are derived from ISS, RTL simulation, and FPGA emulation. It is possible to apply this method to final

ASIC implementations for improved testing. We have implemented a generic framework in which a processor core can be embedded. The framework employs our approach to perform tracing.

A validation environment is used to emit meaningful error descriptions and to point the developer to the location of the error that caused the inconsistency.

Current work focuses on the cycle accurate integration of internal states of hardware accelerators into our consistency check.

Acknowledgement

Substantial parts of the research described in this paper were funded by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung — BMBF) registered there under grant numbers 01BU0661 (MxMobile) and 01BU0643 (Easy-C).

References

1. Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., Barros, E.: The ArchC architecture description language and tools. Int. J. Parallel Program. 33, 453–484 (2005)
2. Burch, J., Dill, D.: Automatic verification of Pipelined Microprocessor Control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994)
3. Chang, Y., Lee, S., Park, I., Kyung, C.: Verification of a microprocessor using real world applications. In: DAC 1999, pp. 181–184 (1999)
4. Gonzalez, R.: Xtensa: A Configurable and Extensible Processor. IEEE Micro., 60–70 (2000)
5. Hosseini, A., Mavroidis, D., Konas, P.: Code generation and analysis for the functional verification of micro processors. In: DAC 1996, pp. 305–310 (1996)
6. Kalte, H., Porrmann, M., Rückert, U.: A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In: Proc. of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip, SoC (2002)
7. Kastens, U., Le, D., Slowik, A., Thies, M.: Feedback Driven Instruction-Set Extension. In: LCTES 2004 (2004)
8. Sawada, J., Hunt, W.: Trace Table Based Approach for Pipeline Microprocessor Verification. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
9. Victor, D., Ludden, J., Peterson, R., Nelson, B., Sharp, W., Hsu, J., Chu, B., Behm, M., Gott, R., Romonosky, A., Farago, A.: Functional verification of the POWER5 microprocessor and POWER5 multiprocessor systems. IBM J. Res. Dev. 49, 541–553 (2005)
10. Yim, J., Hwang, Y., Park, C., Choi, H., Yang, W., Oh, H., Park, I., Kyung, C.-M.: A C-based RTL Design Verification Methodology For Complex Microprocessor. In: DAC 1997, pp. 83–88 (1997)