Proving Functional Correctness of Weakly Programmable IPs - A Case Study with Formal Property Checking

Sacha Loitz, Markus Wedler, Christian Brehm, Timo Vogt, Norbert Wehn, Wolfgang Kunz
Department of Electrical and Computer Engineering,
University of Kaiserslautern, Germany
email: loitz@eit.uni-kl.de.

***Abstract*—In recent years, designing Systems-on-Chip (SoCs) with domain specific and customizable embedded processors (ASIPs) has become standard practice. When compared with general purpose processors on the one hand and dedicated hardwired accelerators on the other hand, these processor cores provide new trade-offs between flexibility, energy and performance. Since they are intended to only run a restricted set of application-specific programs this knowledge is often exploited to further optimize the architecture resulting in weakly programmable IP cores. Such weakly programmable systems raise new challenges for hardware and software verification. The conventional separation of hardware and software verification based on a generic and well-defined instruction set is no longer sustainable. In this paper, we present a case study applying formal property checking to state-of-the-art designs of two weakly programmable IP blocks. A methodology is presented which is oriented at the operations of the ASIP rather than its instructions. As a by-product of our methodology for hardware verification we formalize the software restrictions exploited for optimization of the micro-architecture. We show that an automatic compliance check is feasible which certifies that the software complies with these restrictions. To our best knowledge, this is the first time that functional correctness of ASIP hardware and HW/SW compliance for a realistic design was completely verified using a formal methodology.**

I. Introduction

The use of domain specific and customizable embedded processors has become standard practice in designing complex Systems-on-Chip ([15], [21]). These processors allow for smooth trade-offs of implementation flexibility (in terms of software programmability) against hardware performance (throughput, energy, area). Application specific instructions offered by these processors (denoted as ASIPs) can close the performance gap between traditional processors and dedicated hardwired solutions and improve its energy efficiency. ASIPs offer the benefit of removing application-specific performance bottlenecks by the use of new instructions specific to the application and the adaptation of the datapath pipeline to these new instructions. Some of these ASIP approaches assume a fixed architectural template, e. g., a 5-stage RISC pipeline [2], [24], in which the hardware for the new instructions has to be embedded, others impose no restrictions at all on the pipeline [10] and, thus, give the designer full flexibility in the ASIP design space.

One extreme case within this design space is to remove all flexibility, and thus overhead, in the processor that is not needed by the application and adapting wordsizes, arithmetics, memory hierarchies etc. completely to the needs of the application which yields best performance and energy efficiency. ASIPs designed in this manner share hardly any characteristic of classical RISC pipelines and can rather be considered as application specific building blocks enhanced with programmability. To emphasize this property these special processors are also called *weakly programmable IP (WPIP)*. The design methodology of such processors differs from the traditional top-down ASIP design which starts at C-Code level, performing profiling etc. Instead this approach has a strong emphasis on micro-architecture issues in the datapath pipeline.

In this paper, we consider two challenging ASIPs belonging to this class of weakly programmable IPs. These ASIPs are designed for the outer modem in software defined radio architectures for 3G and 4G wireless applications. The ASIPs were modelled and designed with the ProcessorDesigner tool set from Coware Inc [10]. This tool set provides full flexibility in the ASIP design space imposing no restrictions on the datapath pipeline and memory structure. The cores are used as case studies to evaluate the applicability of formal property checking on highly optimized designs of weakly programmable IP blocks.

Verification of weakly programmable IPs raises a number of new problems and challenges. Since the processor is no longer a pre-verified standard component of the system, and since its architecture and instruction set are perpetually modified during the design process, verification of the ASIP hardware and its software interface consume a great share of the overall design time. Moreover, due to the nature of the ASIP design process, small manual changes to the configuration can have a large impact on the resulting architecture and the probability of introducing errors is high.

Currently, verification of ASIPs is carried out using hardware/software (HW/SW) co-simulation [13], [16]. However, simulation can be very time-consuming and, in some cases, may easily miss bugs. For example, when compared with standard RISC architectures, performance considerations often suggest the use of longer pipelines in ASIPs. The low controllability of the higher pipeline stages makes bug identification by simulation very difficult. Other difficulties result from complex and manifold interactions between a large number of operations as implemented in advanced ASIP pipelines for signal processing applications.

For a reliable proof of functional correctness a methodology

1-4244-2334-7/08/$20.00 © 2008 IEEE

based on formal property checking with a high degree of automation is desirable, as it has become standard practice in the design of ASICs and GPPs [5], [7], [23], [27]. Today, state-of-the-art commercial property checkers can completely verify hardware designs with millions of gates. Such tools have become standard components of industrial design flows.

ASIP design, however, raises significant new challenges for formal verification. Conventional formal verification methodologies for GPPs are typically based on a set of properties which constitute so called refinement relations on the abstract ISA model. For weakly programmable IPs, however, there is no such ISA model. Instructions are implicitely defined by a sequence of operations that can be configured in numerous ways. A single instruction may subsume the behavior of hundreds of classical RISC instructions and, moreover, may depend on the actual configuration of the datapath.

In a conventional HW/SW-system based on GPPs, the software can be verified independently of the hardware. There has been significant progress in formally verifying low-level software, e.g., by the work of [4], [8], [11], [14]. However, note that the conventional separation of hardware and software verification always relies on a a clean programming model as provided by a well-defined instruction set architecture. For weakly programmable IP blocks, however, a clean ISA model is often not available. This imposes new problems and makes it necessary to ensure that the combined system including software and hardware meets its specification.

In [17] an approach to formal property checking for a combined model including software and hardware is proposed. However, this approach requires verification models for both software and hardware to be set up manually. These models are then handled by symbolic verification techniques with the usual restrictions regarding the size of the systems that can be managed. Also, the SAT-based approach proposed in [12] is limited to fairly small programs as it does not abstract from the implementation details of the underlying hardware.

To the best of our knowledge, there is currently no automatic formal verification method available using both, HDL models for the hardware components and C or assembler programs for the software.

In this paper, we present a new methodology based on state-of-the-art (hardware-) property checking and demonstrate its benefit by conducting a case study on the two aforementioned weakly programmable ASIPs. Our methodology addresses the above challenges. In the absence of an ISA-model, the proposed methodology is oriented at the individual operations of the ASIP rather than at its instruction set. We propose a specific style for writing properties which separates the correctness proof for the individual operations and verifying their interplay within instructions. Our methodology results in property checking problems which are within the capacity limits of modern property checkers. As a by-product, our methodology provides restrictions regarding the software to be executed on the ASIP. Often such restrictions are accepted during the design-process as they allow the designer to further optimize the microarchitecture with respect to power and performance. This has motivated us to propose an extension to the conventional model checking technology where such restrictions are captured by an abstracted design model. In the described case study we demonstrate the benefit that can be obtained from this extension.

II. ASIP Design

In this section we provide some background on the ASIP design. One of our case studies is used in the remainder of the paper to explain and motivate the proposed methodology for weakly programmable IP (WPIP) verification. We start with a brief overview of 3G and 4G wireless communication systems in order to demonstrate what requirements our design has to fulfill. Then, we present the instruction set and the structure of the pipeline of the WPIP.

3G and 4G wireless communication systems comprise advanced signal processing algorithms that increase the computational complexity by some orders of magnitude compared to 2G systems. Furthermore, numerous existing and emerging standards require flexible implementations (software defined radio). Hence, some of the signal processing tasks in baseband systems of 3G and 4G communication systems are ideal candidates for WPIP implementations [3], especially channel decoding which is an essential part in the outer modem. Channel coding allows the correction of errors which were induced during the wireless transmission of the signal due to noise. Decoding algorithms used in 3G and 4G are complex iterative algorithms which have no standard signal processing characteristic. Moreover, they use non-standard arithmetics and wordsizes. The various coding parameters and decoding algorithms used in the different standards require a certain flexibility which can not be fulfilled by a hardwired dedicated IP block. Hence channel decoding is a very good candidate for WPIP. Turbo Codes [6] belong to the most efficient channel coding techniques. However, the decoding process is very complex which leads to a high computational effort and large internal data bandwidth. The throughput of Turbo Code decoder implementations on *standard* DSP processors is at least one order of magnitude less than the 3G throughput requirements.

A WPIP for execution of the so called maximum a posteriori algorithm (MAP) [20], [22] was designed. This algorithm is fundamental in Turbo-Code decoding and is also necessary for soft-output decoding of traditional convolutional codes.

The MAP algorithm consists of several loops processing a large data block in a recursive manner in forward and backward direction, respectively. A block can consist of up to 20,000 bits in UMTS. The loop bodies contain some very complex non-standard arithmetic computations. The resulting data path consists of 10 pipeline stages (see Figure 1). Only the fetch (FE) and decode (pipeline DC) stages are common to a traditional RISC pipeline. All the other stages are very specific in order to efficiently perform the individual computations or allowing efficient data access in the MAP loops.

The instruction set comprises dedicated instructions to perform so-called state metric calculations (executed in pipeline stage EX1), so-called forward and backward recursion (executed in pipeline stage EX2), and so called log-likelihood calculations (performed in pipeline stages EX3 and EX4). In

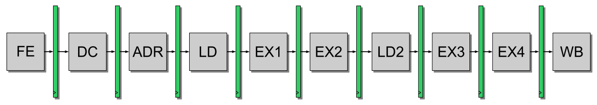

Fig. 1. Dedicated Pipeline of the MAP decoder ASIP

addition, the data path contains special stages for efficient memory access (pipeline stages ADR, LD, LD2). The pipeline is fully optimized for efficient MAP decoding and has a limited programmability compared to a standard RISC pipeline. Thus, this core is a *weakly programmable IP block*. A complete list of the instruction set is summarized in Table I. Each of the instructions listed in this table is far more complex than a conventional RISC instruction. This ASIP is used as fast coprocessor in a wireless modem architecture. Hence, the instruction set does not offer any software defined control instructions. Such an instruction set is typical for WPIPs. The software for a WPIP is written on assembler level to fully exploit the datapath capabilities.

Instr.	Functionality
SMFW	State metric (SM) recursion forwards
SMBW	State metric (SM) recursion backwards
LLRFW	Log-Likelihood recursion forwards
LLRBW	Log-likelihood recursion backwards
AQBW	Acquisition backwards
SADR	Set source address
DADR	Set destination address
SSM0	Initialize state-metric pipeline registers
SM_IO	Load/Store operation on the SM memory
ST_A	Store state-metrics from previous instruction
SMOD	Set the code rate

TABLE I
INSTRUCTION-SET OF THE EXAMINED ASIP

Both WPIPs considered within this paper outperform state-of-the-art DSP implementations for the same application by an order of magnitude in throughput for the decoding algorithms and require substantially less area and energy. [22], [25], [26].

III. FORMAL PROPERTY CHECKING

In this section, we outline a general *verification methodology* for ASIP verification based on property checking. We adapt state-of-the-art formal verification techniques to meet the specific requirements of *weakly programmable IP* (WPIP) designs. Our overall goal is to ensure their functional correctness. In our methodology, this global task is decomposed by the individual instructions and their operations. From the correct functionality of the operations and the correct interplay of the operations considered in each instruction, we conclude functional correctness of the overall design.

Our case study is based on a formal verification technique called *interval property checking* which, in its early versions, was developed at Siemens around the mid 90s. A modern SoC verification environment based on interval property checking is now commercially available by [19].

Interval property checking is a variant of SAT-based property checking and can prove certain temporal logic formulas over the signals of a design. This can be formalized as follows. (We assume that the reader is familiar with standard notions of CTL model checking [9]). Interval properties are CTL* [9] properties of type $\mathsf{AG}\,(p)$, so called *safety properties*, where p only refers to signals of the design within a bounded interval of time, e.g., p can be considered as a Boolean formula over timed signals $\mathsf{X}^t(s)$ which are obtained by applying the generalized X operator to design signals s, with $\mathsf{X}^0(s) = s$ and $\mathsf{X}^t(s) = \mathsf{X}\mathsf{X}^{t-1}(s)$ for $t > 0$. Such an expression p is denoted as *timed boolean expression* throughout this paper. Let t_f, t_l denote the minimum and maximum t used as exponent for the generalized X operator within a timed boolean expression. The time interval $[t_f, t_l]$ will be called *inspection-interval* of a timed boolean expression p and its corresponding interval property $\mathsf{AG}\,(p)$ in the sequel.

Although the set of interval properties is a far less expressive subset of CTL* than other classical temporal logics such as CTL or LTL it has gained significant attention in industrial applications. Interval properties can express the behavior of a design at the register-transfer level in a natural way. Moreover, they can be verified by SAT solving for which highly efficient tools are available. The SAT formulation is based on an iterative circuit model as depicted in Figure 2. Several copies of the transition function are concatenated to generate a combinational model of the circuit under verification.

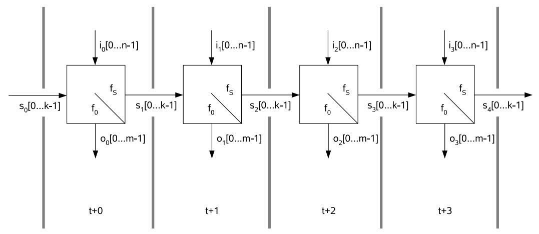

Fig. 2. Iterative circuit model of four time frames

In this model each copy of the transition function is called a *time frame*. Given the inspection-interval $[t_f, t_l]$ of an interval property $\mathsf{AG}\,(p)$ an iterative circuit model of length t_l is conjoined with the boolean expression for p. Finally, a SAT solver is called to check whether the resulting boolean formula is a tautology. In this case, the property $\mathsf{AG}\,(p)$ is proven. However, if the SAT-solver generates a counterexample the user has to examine whether or not the values generated for the state variables s_0 in the first time frame correspond to a reachable state in the original design. If they do not, the counterexample is called *spurious* and we speak of a *false negative*. Otherwise, the property is proven to be wrong.

Fortunately, as a result of the practical verification methodology false negatives do not occur as frequently as it might be expected. Industrial experience shows that they often can be eliminated in an easy way by constraining the state variables at timepoint $t = 0$ by *invariants*, i.e., the property $\mathsf{AG}\,((s_0 \in I) \rightarrow p)$ is proven for a sufficient invariant I. Our intuition explaining this observation is that designers never exactly calculate the reachable states of their overall design and therefore design robustly with respect to global reachability issues. Invariants are state sets which are closed under the image of the transition function and can therefore be proven inductively by the interval properties $\mathsf{AG}\,(reset \rightarrow \mathsf{X}\,(s \in I))$ and $\mathsf{AG}\,(s \in I \rightarrow \mathsf{X}\,(s \in I))$.

Industrial CAD vendors have made a lot of efforts to provide additional language features to support an intuitive specification of interval properties. For example, Table II and Table III show a property that specifies the result of an addition instruction in a simple processor using the property language of the industrial property checker Onespin 360 MV [19] and Accellera's PSL (*Property Specification Language*) [1], respectively. Both properties state that the result of an addition instruction is stored in the result register of the integer pipeline, exactly four clock cycles after the instruction has been decoded.

```
property ADD is
  assume:
    during [t, t+4]: not reset;
    at t: opcode_fetch_reg[10..13] = "0101";
  prove:
    at t+4: result_reg = prev(opnd_A,3) + prev(opnd_B,3);
end property;
```

TABLE II
EXAMPLE OF AN INTERVAL PROPERTY

```
property ADD is
  always
  - - assume part
    next_a[0..4](!reset) &&
     (opcode_fetch_reg[10..13] == "0101")
  − >
  - - prove part
    ( next[4] (result_reg )== opnd_A + opnd_B)
- - end property
;
```

TABLE III
EXAMPLE PROPERTY IN PSL

In these languages **during**[t,t+4]:a and **next_a**[0..4]a, respectively, are shortcuts for the timed Boolean expression

$$\bigwedge_{t=0}^{4} \mathsf{X}^{t}(a).$$

Note, that we may also refer to the past using the temporal operation **prev**. With the relationship $\mathsf{X}^{t}(\mathbf{prev}(x,t))=x$, however, it is always possible to eliminate the **prev** operation from the property. Both examples show the typical structure of interval properties. Properties consist of two main parts. The first part will be called *assumption* in the sequel and describes conditions about the design state and the behavior of the environment within the inspection-interval. The second part is denoted as *commitment* and describes the expected behavior of the design under the *assumption*. This property structure enables the verification engineer to focus on certain aspects of the design in an abstract way. As a result, the selected behaviors can be defined in a simple and compact form so that the resulting properties are easy to review and maintain. By specific methodologies industrial practitioners ensure that the overall property set completely covers the intended behavior of the design. Throughout this paper, we use a pseudocode notation for properties that is close to the Onespin property language. For reasons of space we omit the discussion of the formal semantics of this notation here.

When interval property checking is applied to the design of a general purpose processor the instruction set represents a natural dimension for decomposing the verification task into a set of scenarios. The individual instructions can be executed independently of each other as long as there are no data or control dependencies between them. This results in a methodology where properties are written for each instruction by specifying the effects on the registers that are visible to the programmer and by specifying the effects on the external busses of the WPIP. Table II, as already discussed above, is an example for such a property.

Instructions of WPIPs are far more complex than typical GPP instructions. Moreover, WPIPs allow for reconfiguration of the pipeline via dedicated configuration registers or memory. In this case, instructions of the WPIP may exhibit different behaviors depending on the actual configuration. In order to cope with this situation we split an instruction into individual operations. As an operation we consider a certain sub-function of an instruction that is executed within one specific pipeline stage of the design. Note that this corresponds to the design process of modern ASIP design environments such as the CoWare Processor Designer [18].

In the following, we will sketch how properties are set up for the instructions and operations of a WPIP. We proceed in a top-down manner where we start with properties describing each instruction as a whole. Then, this needs to be refined by specifying the detailed behavior of each instruction as a sequence of operations.

Table IV shows the pseudo-code of a property template used for verifying that the WPIP design correctly implements its instructions. The template is instantiated and customized for every instruction of the ASIP. In this property *instrXYZ_fetched(),instrXYZ_constraint()* and *instrXYZ_performed()* denote timed Boolean expressions. The property states that whenever the instruction *XYZ* is fetched at time t this instruction will be correctly performed starting in the next clock cycle. If certain assumptions must be made about the design state or its environment it is enforced by the template that the verification engineer explicitely formulates all needed assumptions in a timed Boolean expression denoted *instrXYZ_constraint()*. In highly optimized microarchitectures it is possible that fairly complex assumptions have to be made under which an instruction can operate. Note that these assumptions are not only important to describe the possible

```
property instrXYZ is
  assume:
    during [t, t_finished]: no_reset;
    at t: instrXYZ_fetched();
    at t: instrXYZ_constraint();
  prove:
    at t+1: instrXYZ_performed();
end property;
```

TABLE IV
PROPERTY TEMPLATE FOR INSTRUCTION SET VERIFICATION

hardware behaviors, they also represent restrictions for the software that will be executed on this hardware. In the next section, we will outline how these constraints can be re-used to prove that a piece of software targeted to the WPIP complies with the specified restrictions.

When verifying the complete instruction set of a processor the property template of Table IV has to be instantiated for each instruction provided by the hardware. This step can be partially automated. For example, the *instrXYZ_fetched()* macros can be automatically generated using the list of opcodes exported from the ASIP design tool.

Instead of referring to the data sheet of an abstract ISA-model – which is not available for WPIPs – the detailed behavior of an instruction is defined by a sequence of operations. In the following, we show how the specification of an instruction is broken down into individual operations performed by the pipeline. Consider the following timed expression specifying the execution of a backward recursion for state metric calculation by the ASIP presented in the previous section:

SMBW_performed():= decrOffsetSM() $\wedge$ *decrSrcAdr()*
$\wedge$ *loadCV()* $\wedge$ *calculateSMBW()* $\wedge$ *doSMIO)()*
$\wedge$ *decideZeroOverHeadLoop()*

In this example, the *SMBW* instruction is broken down into six operations:

- decrOffsetSM(): decrement the address of the state metric register file.
- decrSrcAdr(): decrement the source address for the next state metric calculation.
- loadCV(): load the so called *channel values* from the address specified by decrSrcAdr() with respect to the selected coderate.
- calculateSMBW(): calculate the 8 state metrics for the loaded channel values using backward traversal. This includes the calculation of the branch metrics in a first step. For the calculation of these 8 state metrics 21 additions and 8 minimum selections on 12 bit words are performed in this operation.
- doSMIO(): store the calculated state metrics at the memory address calculated in decrOffsetSM().
- decideZeroOverHeadLoop(): The SMBW instruction is usually performed several times. For a more efficient implementation a zero overhead loop is executing the instruction for a specified number of iterations. Except for the last iteration of this zero overhead loop the PC is not incremented and the number of iterations that have to be performed is decremented instead.

Breaking down an instruction into several operations has another beneficial aspect. The specifications for the individual operations can be re-used for several instructions. This greatly reduces the verification effort required to completely verify all instructions of the ASIP. Moreover, if the property checker runs into complexity problems a property can easily be decomposed into two properties, each checking a subset of operations. Since all operations are linked by the assumptions and commitments of the properties in a systematic way no verification gap arises from such a decomposition.

IV. Assembler Code Compliance

In the previous section, we have outlined a structured methodology for the specification of a property set that completely verifies the functional correctness of the WPIP hardware. As a by-product of this verification effort a set of constraints has been created that needs to be met by the environment if the design shall function correctly.

In this section, we will outline a fully automatic method to check that software targeted to the WPIP complies with these environment constraints. The set of environment constraints *instrXYZ_constraint()* is denoted by C in the remainder of this paper.

The key idea of the compliance check is to validate whether or not the constraints are valid on every path through the control flow graph of the software. We model the assembler program by its control flow graph $P = (I, E, i_0)$ where I denotes the set of instructions, $E \subset I^2$ denotes the successor relation between instructions and i_0 denotes the first instruction of of the program.

For every constraint in C we know the inspection-interval $[t-a, t+b]$ in which other instructions can depend on the investigated instruction at time t. The size of this interval will be called *dependency depth* of instruction *XYZ*.

For every instruction $i \in I$ used in the program under consideration, we determine the constraint $c_i \in C$ and its inspection-interval $[t-a, t+b]$. We need to prove that the constraint c_i is valid on all paths

$$i_{t-a}, \ldots, i_t = i, \ldots, i_{t+b}$$

in the control flow graph P.

This proof is carried out using the property depicted in Table V. It states that the instructions along the path in the control flow graph comply with the constraint for i. In practice these checks are feasible as the dependency depth of instructions is proportional to the pipeline depth and therefore the number of possible paths to be enumerated remains small. Recall that WPIPs are mainly used for computation rather than control intensive tasks.

```
property instr(i)Complies;
  assume:
    at t-a: instr(i_{t-a})_fetched();
    at t-a+1: instr(i_{t+1-a})_fetched();
    . . .
    at t+b: instr(i_{t+b})_fetched();
  prove:
    at t: instr(i)_constraint();
end property;
```

TABLE V
Property for HW/SW compliance of a single instruction

The efficiency of the compliance check introduced in this section can be further extended by the application of abstraction and black-boxing techniques.

For example, consider the common case that the constraints identified during formal property checking are timed Boolean expressions over the *instrXYZ_fetched()* expressions. Note that these expressions usually compare an internal register of the design with a specific value representing the opcode. An example for this is taken from our case study:

SMBW_fetched() := (instr_fetch_reg == "0111").

In such a case, the property sketched in Table V can be proven using an abstract model of the design that only consists of the register *instr_fetch_reg*. In our experience, an initial abstraction including only the registers used in the above property can be quickly refined towards an abstraction which is sufficient for an efficient compliance check. Usually only a few refinement steps of including logic in the fanin of the registers are required. Such abstractions can be generated automatically and make HW/SW compliance checking tractable even for large designs.

V. EXPERIMENTS

For the experimental evaluation of the proposed methodology we formally verified two state-of-the-art designs which implement algorithms for channel decoding. The first design, MAP, implements a MAP decoder and has been described in Section II. The second design, FlexiTrep, is even more complex than MAP, and provides instructions to support multiple channel decoding algorithms. It consists of 15 pipeline stages, a sophisticated distributed memory architecture within the pipeline stages and a dynamically reconfigurable channel code control unit which supports multi-context instructions, fast context switches between different codes and efficient operand management. Detailed information about FlexiTrep can be found in [26].

The verification methodology proposed in this paper has been applied to both designs. Specifically, the functionality of MAP was covered by 28 properties which were set up based on the templates presented in the previous section. The proof of the entire property set takes a total CPU-time of 67 s. Most of this time is spent in the verification of the combined recursive state metric calculation and the log-likelihood calculation. All other operations have been verified in less than 1 s of CPU-time. All experiments were conducted using the industrial property checker Onespin 360 MV[19]. The machine for the experiments was running SUSE Linux 10.1. on an Intel dual core CPU with 2GB RAM.

The FlexiTrep WPIP design features 38 instructions to support software implementations of multiple channel decoding algorithms. Some of the instructions perform operations in all of the 15 pipeline stages. Such instructions correspond to hundreds of RISC instructions. For complexity reasons, some of the properties verifying the most complex instructions had to be split in up to three properties. The final property set therefore consists of 42 properties.

The verification of the entire property set takes about 100 minutes of CPU-time. The time required for the individual properties ranges from a few seconds for simple initialization instructions up to 20 minutes for some of the instructions performing state metric calculations.

Note that the CPU-times required for formal verification of the complete property sets are competitive even when compared with simulation-based verification.

Both WPIPs have been simulated intensively using the debugger of the ASIP design tool and had been signed off for industrial application. Yet, in both cases serious bugs could be identified after applying our formal methodology.

In the MAP design, two bugs were identified that were caused by inconsistent bit widths. Two eight bit numbers had to be added into a nine bit result. The generated hardware performs an eight bit addition and sign extends the result afterwards instead of first extending the operands and performing the addition on the extended operands.

For the FlexiTrep design we were able to show that the saturation operation located in pipeline stage 14 did not work correctly under certain conditions on the operands. More precisely, for numbers smaller than -64, two out of three saturation units checked a wrong saturation condition. During the intensive sign-off simulations this bug was missed as it only occurs in very rare cases and can only be observed at the end of the pipeline. Due to the limited controllability of deep pipeline stages it is hard to force a simulator into such corner cases.

Another bug identified by property checking turned out to be caused by a late code change where the designer forgot to remove a specific value assignment to a control signal. The value of this signal was now calculated twice causing a race condition in the generated RTL code. In the design environment, however, this race condition had not been visible.

In a second step, for both designs the compliance of the software with the developed hardware has been checked using the method described in Section IV.

Within the MAP design certain power optimizations caused a data dependency between the instructions for forward and backward calculations. It was deliberately left to the software developer to solve the resulting hazards by inserting NOP instructions between the code fragments working in different directions. The compliance checker disclosed, however, that this rule had not always been obeyed within the tested software. More precisely, we tested eight programs differing in the direction of calculation and the number of bits to be processed. It turned out that three out of four programs starting with the backward calculation failed the compliance check.

By contrast, the FlexiTrep WPIP offers a stall unit to cope with such problems. Therefore, the programmer is relieved from taking care of dependencies between instructions. Nevertheless, also here the compliance check proved quite beneficial. We set up a verification experiment where we assumed that the stall unit is deactivated and we attempted to prove HW/SW compliance under this condition. Not surprisingly, counterexamples were generated. The compliance checker explicitely pinpoints the individual instructions of the assembler code that cause the problem and visualizes the instructions involved in the conflict. This provides the software engineer with valuable information when manually optimizing his code. With a few modifications of the software, e.g., adding NOPs at these positions, the conflicts could be removed. Moreover, we identified two programs where the compliance checker has proven that no stalls will ever happen and thus the stall unit could be turned off.

The results of the compliance checks are summarized in

Table VI. The table is organized as follows. The first two columns show the name of the program and the number of instructions to be analyzed by the compliance checker. The CPU-time and the result of the compliance checks are indicated in columns three and four.

Program	#Instr	CPU-time	Result
fw_287	221	1.54 s	Fails
fw_288_1	230	1.42 s	Holds
fw_288_2	221	1.48 s	Fails
fw_289	236	1.54 s	Fails
bw_287	225	1.31 s	Holds
bw_288	225	1.33 s	Holds
bw_289	245	1.35 s	Holds
DuoBinary AP	42	2.34 s	Holds
DuoBinaryS16 AP	43	2.3 s	Holds
RSC NT IL	99	3.94 s	Fails
RSC relSMA	302	10.9 s	Fails
VA 133	99	3.86 s	Fails
VA 23 fast	61	2.63 s	Fails
VA 557	289	10.38 s	Fails
VA RSC 133	91	3.55 s	Fails

TABLE VI
REQUIRED TIMES FOR THE COMPLIANCE CHECK

VI. CONCLUSION

This paper presents a formal verification methodology which adapts property checking to the specific requirements of *weakly programmable IP* (WPIP) designs. As a by-product of the steps taken to verify the hardware we obtain a formal specification for the restrictions that the software must comply with when running on this WPIP. We demonstrate that complete functional hardware verification of the WPIP as well as a fully automatic compliance check for the software is feasible by making only small extensions to a state-of-the-art formal property checker.

REFERENCES

[1] Accellera Organization Inc. Property specification language - reference manual, version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 9 2004.

[2] ARC Inc. www.arc.com/company/index.html.

[3] G. Ascheid and H. Meyr. Opportunities for Application-Specific Processors: The Case of Wireless Communication. in *Customizable Embedded Processors*, pages 11–37, Morgan Kaufmann Publishers, 2007.

[4] T. Ball and S. K. Rajamani. The Slam Project: Debugging System Software via Static Analysis. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Portland,, January 2002.

[5] S. Berezin, E. M. Clarke, A. Biere, and Y. Zhu. Verification of Out-Of-Order Processor Designs Using Model Checking and a Light-Weight Completion Function. *Formal Methods in System Design*, 20(2):159–186, 2002.

[6] C. Berrou and A. Glavieux. Near Optimum Error Correcting Coding and Decoding. *IEEE Transactions on Communications*, 44(10):1261–1271, Oct. 1996.

[7] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying Safety Properties Of A Power PC-TM Microprocessor Using Symbolic Model Checking Without BDDs. In *Proc. International Conference on Computer Aided Verification (CAV)*, 1999.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, London, England, 1999.

[10] Co-Ware Inc. www.coware.com.

[11] P. Godefroid. Software Model Checking The Verisoft Approach. *Formal Methods in System Design, 2005*, 26:77 – 101, 2005.

[12] D. Große, U. Köhne, and R. Drechsler. HW/SW Co-Verification of Embedded Systems using Bounded Model Checking. In *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*, pages 43–48, 2006.

[13] D. Harris, D. Stokes, and R. Klein. An RTOS On Simulated Hardware Using Co-Verification. In *Embedded Systems Conference*, 2000.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy Abstraction. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[15] P. Ienne and R. Leupers. *Customizable Embedded Processors*. Morgan Kaufmann Publishers, 2007.

[16] R. Klein. Hardware/Software Co-Verification. In *Embedded Systems Conference 2003*, 2003.

[17] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Combining software and hardware verification techniques. *Formal Methods in System Design*, 21:251 – 280, 2002.

[18] Co-Ware Processor Designer. coware.com/products/processordesigner.php .

[19] Onespin Solutions GmbH Munich, Germany. www.onespin.com.

[20] P. Robertson, P. Hoeher, and E. Villebrun. Optimal and Sub-Optimal Maximum a Posteriori Algorithms Suitable for Turbo Decoding. *European Transactions on Telecommunications (ETT)*, 8(2):119–125, March–April 1997.

[21] C. Rowen. *Engineering the Complex SOC. Fast, Flexible Design with Configurable Processors.* Prentice Hall PTR, 2004.

[22] D. Schmidt and N. Wehn. Hardware/Software Tradeoffs for Advanced 3G Channel Decoding. in *Customizable Embedded Processors*, pages 361–379, Morgan Kaufmann Publishers 2007.

[23] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, McConville, and T. Swanson. Verification of the Cell Broadband Engine (TM) Processor. In *Proc. International Design Automation Conference (DAC)*, pages 338 – 343, 2006.

[24] Tensilica Inc. www.tensilica.com.

[25] T.Vogt and N.Wehn. A Reconfigurable Application Specific Instruction Set Processor for Viterbi and Log-MAP Decoding. In *Proc. IEEE Workshop on Signal Processing (SIPS'06)*, pages 142–147, Banff, Canada, October 2006.

[26] T. Vogt and N. Wehn. A reconfigurable application specific instruction set processor for convolutional and turbo decoding in a sdr environment. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, 2008 (to appear).

[27] K. Winkelmann, D. Stoffel, G. Fey, and H. Trylus. Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. In *Proc. International Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004.