Automated Formal Verification of Processors Based on Architectural Models

Ulrich Kühne*
LSV ENS de Cachan
Cachan, France

Sven Beyer†
OneSpin Solutions GmbH
Munich, Germany

Jörg Bormann†
Abstract RT Solutions GmbH
Munich, Germany

John Barstow
Infineon Technologies Ltd
Bristol, UK

***Abstract*—**
To keep up with the growing complexity of digital systems, high level models are used in the design process. In today's processor design, a comprehensive tool chain can be built automatically from architectural or transaction level models, but disregarding formal verification. We present an approach to automatically generate a complete property suite from an architecture description, that can be used to formally verify a register transfer level (RTL) implementation of a processor. The property suite is complete by construction, i.e. an exhaustive verification of all the functionality of the processor is ensured by the method. It allows for the efficient verification of single pipeline processors, including several advanced processor features like multicycle instructions. At the same time, the structured approach reduces the effort for verification significantly compared to a manual complete formal verification. The presented techniques have been implemented in the tool FISACo, which is demonstrated on an industrial processor.

I. Introduction

The complexity of digital hardware systems has shown an exponential growth over the last decades and it is growing still. To keep track of large systems during the design process, high level models are used increasingly. Especially for the design of processors, architecture or transaction level models form the core of an elaborate tool chain that enables the automatic generation of simulators, assemblers or compilers, like *Facile* [27] or *LISA* [9]. However, formal verification of the functionality of the design is still not part of this tool chain.

There exist several techniques for the verification of hardware designs. In simulation based verification, the outputs of the implementation are compared to a golden reference model, that is usually based on a transaction level description. But, simulation is not well suited to cover the whole functionality of a pipelined processor because achieving a sufficient design quality for such a processor requires a huge simulation-based verification effort and there is no guarantee that all possible bugs have been considered. In contrast, formal techniques offer the highest quality of verification [15].

One successful technique is *Interval Property Checking* (IPC) [23], a technique similar to Bounded Model Checking [3]. IPC is used to check if a system satisfies a set of properties about the operations of a design like the processing of a request in a bus bridge, the execution of an instruction in a processor pipeline, or an arbitration cycle in an arbiter. IPC has been extended with further proofs which ensure that a set of properties verifies all input/output behavior of a circuit [5]. This methodology has already been used in industrial context for the verification of a wide variety of designs [12] including small or medium size processors [6].

However, these projects also illustrate that the integration of a thorough formal examination into industrial verification practice requires larger changes to the education and opinions of verification engineers. Compared to simulation based approaches, formal verification requires a deep knowledge of the internals of the design under verification (DUV) in order to write assertions. An important motivation of the work summarized here and presented in [20] is therefore, that the automation of the formal verification of some well defined class of circuits eases the migration from simulation to formal verification and hence helps to introduce this technology. We chose smaller single pipeline processors as this class.

For processors, a structured manual verification flow is available today [2]. But, automation of the verification is quite low, the more comprehensive the verification is. On the other hand, existing approaches for the automatic verification of processors (see related work in Sect. II) often require a background of deep and general insight into verification goals and correctness criteria.

In this paper we present a technique for the automatic generation of a complete property suite for processors. The starting point of the approach is an architecture description of the processor. By defining a number of mapping functions the user captures how the abstract concepts are mapped to the register transfer level (RTL) implementation. These mapping functions refer to pipeline stages, stall and cancel signals, and similar objects that design and verification engineers are familiar with. Following this approach, the specification is captured in a concise and readable form, while the underlying general processor model enables the verification of several advanced processor features like multicycle instructions, out-of-order termination as well as exceptions. The generated property suite is complete by construction in the sense of [5]. As a driving verification engine, the OneSpin 360 MV tool [24] is used, offering the performance and capacity to formally verify whole processor designs.

The main contribution of this work is a well structured yet pragmatic approach to tackle the formal verification of processors. It offers an exhaustive verification for a certain class

This work was carried out jointly by the group for computer architecture at University of Bremen*, Germany and OneSpin Solutions GmbH†. It was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerisoftXT under contract no. 01 IS 07008 C.

of designs, while the automatic generation of the properties increases the verification productivity significantly compared to manual coding of properties. As the input for the approach is an abstract architecture description, the method can easily be integrated with existing tool chains for processor design. The automatic generation of the properties is implemented in the tool FISACo. The approach is demonstrated with an industrial control processor used in embedded automotive systems, a domain with particularly high quality requirements.

The paper is structured as follows. Related work is discussed in Sect. II. In Sect. III, our formal verification techniques are reviewed. The automatic property generation is described in Sect. IV. Sect. V shows the application of the approach to an industrial processor design. Sect. VI concludes the work.

II. Related Work

An approach for the automatic equivalence verification of general transaction level models (TLM) with timed implementations is presented in [4], where the different levels of abstraction are related by events. However, the lowest level of abstraction in [4] is behavioral RTL and it is not clear how the concept of events relates to optimized pipeline designs. In other words, an automated equivalence check between a sequential processor architecture and a pipelined RTL implementation is not feasible for optimized industrial designs.

There has been work on the formalization of pipelined designs. Part of the approaches in the literature use formal models for the automatic design of correct pipelines [19], or to accompany the design process [10], [16]. In [10], starting from a simple model, the design is incrementally refined until a pipelined implementation is obtained. A CTL specification is transformed along with the design to prove the correctness of the refinement steps. A similar approach is presented in [22]. It decomposes the correctness proof for a complex pipelined machine with branch prediction into several steps, the first of which proves the compliance of a simple version of the processor with its ISA. The drawback of these approaches is that they cannot handle industrial designs containing legacy code and manual optimizations that are needed to match hard power and timing constraints.

There are various techniques for the verification of existing processors [1], [13], [14], [30]. In [1], a formal pipeline model is introduced that is based on *parcels* (instructions) processing through the pipeline. By instantiating several predicates describing the pipeline, the correctness of the design can be proved formally. However, the model is rather abstract and the predicates seem difficult to derive. In contrast, we provide a clear distinction between the architecture layer and the mapping to the implementation. Furthermore, our mapping functions have a more intuitive counterpart in the designer's intent of implementing a pipeline.

Further approaches for processor verification rely on interactive theorem proving [18], [26], [29]. This generally offers a high level view on the design. Theorem proving however requires a significant level of expertise that is usually not available to designers or verification engineers in practice.

Approaches for the automatic generation of properties are given in [17], [25]; they are based on learning dependencies or properties from simulation traces. However, they are only suited for an initial design understanding rather than for a verification against a specification. In contrast, our approach starts with a specification that is then related to the implementation in a well structured way.

III. Formal Verification Setting

Within the last two decades, there has been a lot of research in formal verification techniques. Methods based on Boolean satisfiability (SAT) have proven to be a robust solution. One prominent technique is SAT based *Bounded Model Checking* (BMC), that has first been described in [3]. Successive improvements in performance have made BMC a suitable method for the formal verification of larger scale designs. For the work at hand, we use the techniques described in [23], referred to as *interval property checking* (IPC). In the following, this verification methodology will be briefly outlined.

In contrast to BMC, only safety properties are verified using IPC. As digital circuits always have a finite response time, this is not a serious restriction in practice. It is rather natural to capture the specification of a design in terms of safety properties. Furthermore, using IPC, these properties can be verified with bounded proofs, which can be checked efficiently using a SAT solver.

The main idea of IPC is to use an arbitrary starting state instead of the initial state used in BMC. Any property that holds starting from an arbitrary state then also holds from any reachable state and thus, it is exhaustively verified. Conversely, false negatives can occur in IPC, i.e. counterexamples for properties starting in unreachable states may be produced. These false negatives need to be removed by adding invariants in order to restrict the starting state. For more details on the idea of IPC and the following formalization, refer to [23].

A synchronous circuit is modeled as a finite state machine (FSM) $M = (I, S, S_0, \Delta, \Lambda, O)$ with input alphabet $I \subseteq \mathbb{B}^n$, output alphabet $O \subseteq \mathbb{B}^w$, a finite set of states $S \subseteq \mathbb{B}^m$, output function Λ and next state function Δ. The set $S_0 \subseteq S$ denotes the initial states. With next state function $\Delta : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}^m$, the transition relation of the circuit is given by

$$T(s, s') = \exists x \in \mathbb{B}^n : s' \equiv \Delta(x, s). \tag{1}$$

A safety property $f = \mathsf{AG}(\varphi)$ is translated to a Boolean function $[[f]]_t$, checking the validity of formula φ at timepoint t. Here, the translation is done such that a satisfying assignment of $[[f]]_t$ corresponds to a counterexample of φ. The resulting function depends on the inputs, outputs and states within a bounded time interval $[0, c]$. IPC searches for counterexamples by solving the SAT instance

$$\bigwedge_{i=0}^{c} T(s^{t+i}, s^{t+i+1}) \wedge [[f]]_t. \tag{2}$$

The transition relation is unrolled within the time interval $[0, c]$ and it is connected to the single instantiation of $[[f]]_t$. In order to avoid unreachable counterexamples, invariants are added. In many cases, such invariants can even be generated automatically [31]. In the context of the described methodology, the needed invariants are usually less complex than

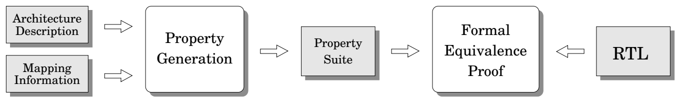


Fig. 1. Verification Flow with Property Generation

the main properties; they can thus be verified using inductive proof techniques like *k-induction* [28]. For more details on the method, refer to [23].

IPC is a powerful verification technique, enabling the formalization of a specification in terms of safety properties and its verification against the implementation. However, to be sure that no bugs have been missed, the verification engineer needs to reason about the *completeness* of the written property suite. A technique to formally check whether a set of properties forms a complete specification is described in [5], [8]. These techniques have been successfully applied to industrial processor designs [6].

Completeness analysis determines whether every possible input scenario—corresponding to a transaction sequence of the design—can be covered by a chain of properties that predicts the value of states and outputs at every point in time. In other words, any two designs fulfilling all the properties of a complete property suite are formally equivalent. The completeness analysis basically boils down to check in the end state of each property whether (1) there is always a successor property with matching assumptions, (2) the successor property is uniquely determined and (3) each property describes the outputs and states of the design uniquely. For more details on the methodology please refer to [5], [8].

For the formal verification of the generated property suite against the RTL, we use OneSpin 360MV [24]. This commercial solution covers the required spectrum of formal verification—from the verification of SystemVerilog assertions all the way to the automatic completeness analysis described above. Among various other proof engines, 360MV also offers IPC and k-induction with sufficient capacity and performance to handle the complete verification of processors [6].

IV. Verification Using Generated Properties

Technically, the basis of the approach presented here is to provide a general formal processor model that can be customized by the user to match his specific implementation. The general processor model can be thought of as a tool box with several design features to be picked out. The customization is done by setting the architecture design parameters, like the number of pipeline stages and the possible interface transactions. Furthermore the mapping from the architecture description to the RTL has to be established by defining a number of mapping functions. The basic flow is shown in Fig. 1. The general processor model consists of three parts:

1) The **pipeline model** describes the movement of the instructions through the stages
2) The **datapath model** describes register access and data forwarding
3) The **interface model** describes memory and bus accesses

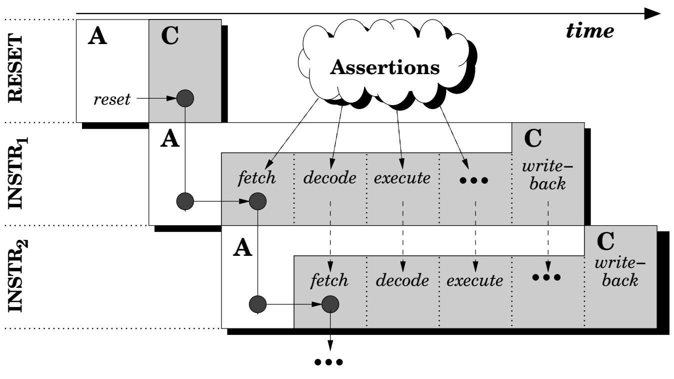


Fig. 2. Interaction of generated properties

After identifying the visible registers and interfaces, the instruction set and the exception behavior of the processor are described on the architecture level. The generated property suite consists of *instruction properties* and a set of *consistency assertions*. While each instruction property describes the processing of a single instruction until it leaves the pipeline, the consistency assertions ensure the correct interaction of multiple instructions and the consistent pipeline behavior, if no instruction is present in a dedicated stage. The latter includes e.g. checking that empty stages will not update any state elements. These assertions also help the user in finding an appropriate mapping by giving him a feedback for debugging.

Basically, the equivalence of the property suite and the DUV is established by chaining the generated properties, as shown in Figure 2. Each property is depicted as a rectangular box, consisting of an assume part (*assumption A*) and prove part (*consequent C*, shaded gray in the figure). The properties are hooked up at the time point when the processor is ready to execute the next instruction, represented by the big black dots in the picture. Thus, starting from reset, the first property proves that the *new instruction state* (NIS) will be reached. Then, the following properties assume the NIS and prove that after fetching the dedicated instruction, NIS will be reached again, enabling the connection to the next instruction property.

The basic approach has been described in detail in [20]; it is based on a patent application [7].

A. General Processor Model

The approach presented here is limited to a class of processors that is common in industrial designs. The class is characterized by the following features:

- Single pipeline
- In-order-execution, out-of-order termination
- Register files with multiple prioritized write channels
- Exceptions and interrupts
- Delayed branch instructions
- Branch prediction
- Multicycle instructions
- Multiple interfaces, including pipelined protocols

Note that a typical CPU also contains complex data memory and prefetch logic. With our approach, the core of such a CPU can be verified with generated properties, providing exact interface descriptions to the data memory and prefetch.

These modules can in turn be verified manually, thus ensuring correctness of the overall CPU. In such manual extensions, the already established mappings and models can easily be reused.

The components that are included in the architecture view are described in the following. A processor receives its instructions via an instruction memory interface, that is addressed by a program counter PC. The currently processed instruction word is denoted IW. There can be an arbitrary number of architecture registers and flags. There can be interfaces to data memories or buses, each associated with a set of transactions, at least containing the idle transaction.

The instructions are described on the architecture level. In order to verify them against the RTL implementation, a mapping has to be established by the user. For the components PC and IW, the corresponding mapping function is usually pointing to a dedicated signal in the RTL showing the value of the program counter and an instruction register, respectively. However, the mapping of an architecture register file requires a model of the pipelining due to forwarding mechanisms that are part of every pipelined processor design. We first introduce the architecture description, followed by a discussion of the models for the pipeline, the data path, and the interfaces. Finally, the generation of the property suite is described, and the completeness of the model is discussed.

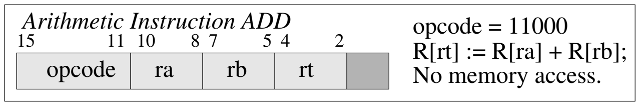
B. Architecture Description

In our approach, there is a clear distinction between the architecture description and the mapping functions. In this way, a readable and proven correct description of the ISA is obtained. The mapping functions relate the ISA to the RTL.

In the first section of the architecture description, the components of the processor are listed, comprising all architecture registers and flags. Furthermore the interfaces to memories and buses are given, as well as the respective transaction types on these interfaces. The main section of the architecture description consists of the ISA description, where all instructions of the processor are defined. In the ISA description, the registers are referred to by their specification name.

For each instruction, first the execution condition is given (`TRIGGER`). Then, the updates of the program counter and the architecture registers and flags need to be defined, followed by the definition of one transaction per interface. The updates are defined by the read registers (`VREGISTER`), the target register (`UREGISTER`) and the value that will be written by the instruction (`UPDATE`).

As an example, consider Fig. 3(b) with a simple processor description including an `ADD` instruction. The triggers for the instruction are divided into two statements, one of which only depends on the architecture state (`TRIGGER_STATE`), while the second one depends on the instruction word (`TRIGGER_IW`). Besides the update of the program counter in line 8, there is one update of the register `R`, where two registers are read addressed by parts of the instruction word (lines 9 and 10). The target register is given in line 11 and the sum of the two source registers is defined in line 12. Finally, there is no transaction on the data memory interface, indicated by the statement `DMEM_IDLE` in line 13.



(a) Specification

```
 1  registers  := R;
 2  interfaces := DMEM;
 3  transactions_DMEM := IDLE, READ, WRITE;
 4
 5  simple_instruction ADD {
 6    TRIGGER_STATE := true;
 7    TRIGGER_IW := IW[15:11] == ADD_op;
 8    UPDATE_PC := (PC + 2)[7:0];
 9    VREGISTER_1 := R(IW[10:8]);
10    VREGISTER_2 := R(IW[7:5]);
11    UREGISTER_1 := R(IW[4:2]);
12    UPDATE_1 := (VREGISTER_1 + VREGISTER_2)[15:0];
13    DMEM_IDLE; }
```

(b) Architecture description

Fig. 3. Informal specification and architecture description example

C. Pipeline Model

In a pipeline, the processing of instructions is overlapped in order to speed up computations. Thus, a new instruction starts before the preceding one has terminated. For example, a typical simple pipeline would partition an instruction into fetching the instruction word from the memory, decoding it, executing logical and arithmetic operations and writing the result back into the register file. Note that this section only introduces basic pipeline modeling for the control path in order to keep track of the different instructions in the pipeline. The handling of forwarding is part of the data path of a pipeline and discussed in the following Sect. IV-D.

The major challenge in designing a correct pipeline are hazards, i.e. conflicts between instructions that are processed at the same time in different stages. If an instruction needs data that is currently being computed by a preceding instruction, a read-after-write conflict occurs and the succeeding instruction needs to wait for the data. Thus, a mechanism to *stall* a stage is needed. Another hazard is related to branching instructions. When a jump is taken, this is typically detected at a time when subsequent instructions from the sequential program flow already have been fetched. Therefore, the pipeline must possibly be cleaned from wrongly fetched instructions, requiring a *cancel* mechanism. As this may lead to stages that are not processing any instructions, it is desirable to distinguish between empty and *full* stages to prevent spurious register updates or similar faults. Based on these requirements, we now define our pipeline model.

Given the number of pipeline stages n, we define the set $\mathcal{S} = \{1, 2, \dots n\}$ of pipeline stages. The pipeline architecture is further classified by defining some constants that refer to certain stages like the decode stage $dec \in \mathcal{S}$ and the stages $ia, iv \in \mathcal{S}$ that denote the stage when the instruction memory is accessed and when the instruction word is valid, respectively. The processing of instructions by the pipeline is defined by the mapping functions[1] $full, stall, cancel : \mathcal{S} \rightarrow \mathbb{B}$.

The value of $full(s)$ reflects if the pipeline stage s currently

[1]The state of the design is an *implicit* parameter of all mapping functions.

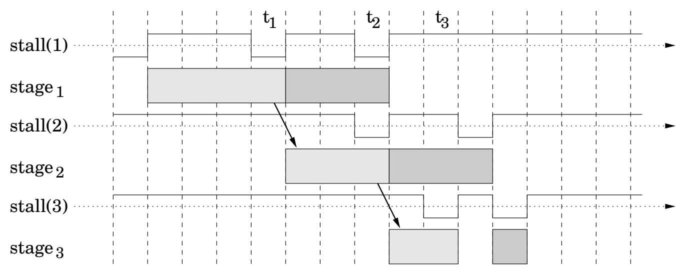


Fig. 4. Normal processing of instructions by the pipeline

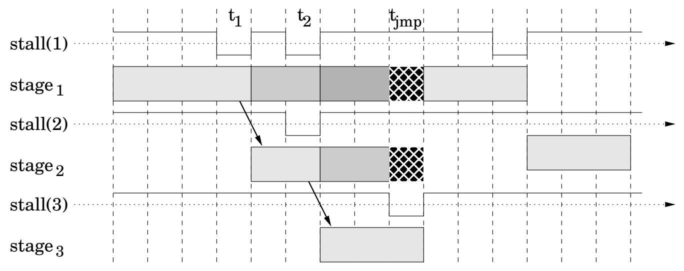


Fig. 5. Pipeline for a taken jump instruction

holds an instruction. If $stall(s)$ is true, the instruction in stage s will not proceed to the succeeding stage $s+1$. $cancel(s)$ indicates that the instruction currently in stage s will be removed from the pipeline and will have no more effects in later stages. The normal processing of two consecutive instructions is shown in Fig. 4, where time progresses from left to right. The time-points when the first instruction is allowed to proceed to the next stage are denoted by t_1 to t_3, i.e., $stall(s)$ evaluates to false at t_s. The boxes indicate the time-points when the respective stage is processing an instruction, i.e. $full(s) = 1$. The pipeline for a taken jump or mispredicted branch instruction is shown in Fig. 5. At timepoint t_{jmp}, the canceling of two succeeding instructions is indicated by the dark boxes. After the taken jump, the target instruction is fetched from the instruction memory.

The mapping functions have to be defined by the user. This means for example, that the user needs to identify how the implementation encodes the fact that a stage is full. Since the functions are used in the properties, the verification fails as long as the model is not completed properly.

In addition to the basic model, further pipeline operations can be supported. It is common for instructions to leave the pipeline before the last stage, if no more actions will be taken in later stages, in order to prevent conflicts. Thus, a *last stage* can be defined for each instruction. Exceptions are a crucial feature for practical applications. By nature, they interrupt the normal instruction processing. The most general exception model, that is still suited to conform with our approach, is an *injection* of a new instruction into the pipeline after an exception has been acknowledged. Finally, for more complex arithmetic operations or interactions with protocol driven interfaces, multicycle instructions are frequently used in processor designs. Typically, an FSM in an early stage is responsible for *dispatching* partial instructions in the pipeline.

For these refinements of the simple model, additional map-

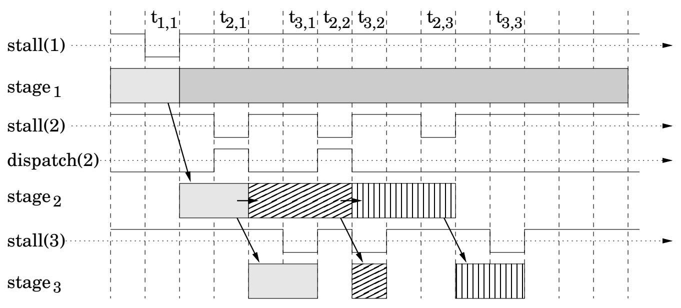


Fig. 6. Pipeline for a multicycle instruction

ping functions for of out-of-order termination, exceptions, and multicycle instructions need to be defined.

$$last\_stage, inject, dispatch : \mathcal{S} \to \mathbb{B}, \tag{3}$$

Here, $last\_stage(s)$ indicates that the instruction in stage s will leave the pipeline, $inject(s)$ states that an instruction will be injected into stage s in the next cycle due to an exception, and $dispatch(s)$ describes that a multicycle instruction is started in stage s. The pipeline of a multicycle instruction according to our model is shown in Fig. 6. There, the partial instructions are dispatched in stage 2.

D. Data Path Model

Based on the above control path model of the pipeline, we can now define the data path model, describing the way how data is read, forwarded and stored in the registers.

For a register file R, a mapping function $current_R : \mathcal{I}_R \to \mathcal{D}_R$ is defined that returns the current implementation state of the register, where $\mathcal{I}_R$ is the index set and $\mathcal{D}_R$ is the data domain of register R. For the data path of the register the following mapping functions have to be defined:

$$write_R, valid_R : \mathcal{S} \to \mathbb{B} \tag{4a}$$
$$dest_R : \mathcal{S} \to \mathcal{I}_R \tag{4b}$$
$$data_R : \mathcal{S} \to \mathcal{D}_R, \tag{4c}$$

where $write_R(s)$ indicates if the instruction in stage s is going to update register R, while $dest_R(s)$ and $data_R(s)$ specify the target register and the data to be written, respectively. By $valid_R(s)$, it is stated if stage s already produces a valid result. With these building blocks, the forwarding in the pipeline to some forwarding target stage $s \in \mathcal{S}$ can easily be captured: the value of a register R with index $i \in \mathcal{I}_R$ in the forwarding target stage s is recursively given by checking whether succeeding stages write to register i; this corresponds to the forwarding logic in the pipeline.

$$Data_R(s,i) = \begin{cases} current_R(i), & \text{if } s \geq writeback_R; \\ data_R(s+1), & \text{if } write_R(s+1) \wedge \\ & \quad dest_R(s+1) = i; \\ Data_R(s+1,i), & \text{otherwise.} \end{cases}$$

Note that this automatically generated function $Data_R$ actually captures the complex mapping of the visible register R to the implementation, i.e., the architecture value of R for an instruction in the pipeline is the value of $Data_R$ in the forwarding target stage of that instruction. Since the value of $Data_R$ may be invalid because the result of some instruction is not available yet, we introduce an additional mapping function capturing whether the forwarding data is indeed valid:

$$Valid_R(s,i) = \begin{cases} false, & \text{if } s < dec; \\ true, & \text{if } s \geq writeback_R; \\ valid_R(s+1), & \text{if } write_R(s+1) \wedge \\ & dest_R(s+1) = i; \\ Valid_R(s+1,i), & \text{otherwise.} \end{cases}$$

E. Interface Transactions

In order to verify the interfaces of the processor, the following model is used. For each interface IF the constants $da_{IF}, dv_{IF} \in \mathcal{S}$ denote the stage when a data access is issued and when valid data is returned, respectively. For each interface, a set of transactions TA_{IF} is defined by the user with at least $IDLE \in TA_{IF}$. For each transaction $ta \in TA_{IF}$ a function $ta_{IF} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$ is defined, where $ta_{IF}(addr, wdata)$ captures that the specified transaction takes place in the design, optionally involving the address $addr$ and (for writing transactions) the write data $wdata$. As for the example in Fig. 3(b), the three functions $IDLE_{DMEM}, READ_{DMEM}$ and $WRITE_{DMEM}$ need to be defined, capturing for given address and data, if the respective transaction is issued on the data memory interface.

Besides the transactions, for each interface a mapping function $rdata_{IF}$ points to the implementation port, where data is read in to the processor. Finally, there is a static interface to the instruction memory, given by the predicate $ibus\_read : \mathbb{N} \rightarrow \mathbb{B}$, which checks if the instruction memory is currently being accessed for a given value of the program counter.

F. Consistency Assertions

While the above models describe the processing of instructions by the successive pipeline stages, additional assertions are needed for the overall correctness of the processor. This includes the behavior of empty pipeline stages as well as the interaction of succeeding instructions. For this purpose, a set of *consistency assertions* are automatically generated.

Note that the overall verification is fail safe, i.e. it cannot succeed if the design is not correct. But, even for a correct design, finding the appropriate mapping functions can be difficult. The consistency assertions provide useful information on the status of the modeling. Failing assertions can point the user to certain mapping functions that need to be revised to complete the verification, thereby guiding the debugging process.

We show the following assertion as an example. For a more detailed description of the consistency assertions, see [20].

$$\begin{aligned} &\forall s, 2 \leq s \leq n : \\ &(\ (\neg full^t(s-1) \vee stall^t(s-1)) \wedge \\ &(\neg full^t(s) \vee \neg stall^t(s))\ ) \Rightarrow \neg full^{t+1}(s) \end{aligned} \tag{5}$$

This assertion states that it is illegal to create full stages in the middle of the pipeline: when the stage before s is empty or stalled, and s is empty or it will proceed to the next stage, then s must be empty in the next cycle. Here, f^t denotes the value of f at timepoint t. Other assertions ensure, for example, that instructions do not overwrite each other and that empty pipeline stages do not have an effect on the visible registers or issue interface transactions.

TABLE I
USER INPUT FOR PROPERTY GENERATION

(a) Constants

Name	Domain	Description
n	$\mathbb{N}$	number of stages
dec	$\mathcal{S} = \{1, \ldots, n\}$	decode stage
ia	$\mathcal{S}$	instruction memory access stage
iv	$\mathcal{S}$	stage in which instr. word is valid
int	$\mathcal{S}$	highest stage for interrupt injection
da_{IF}	$\mathcal{S}$	access stage for interface IF
dv_{IF}	$\mathcal{S}$	data valid stage for interface IF
$writeback_R$	$\mathcal{S}$	writeback stage for register R

(b) Mapping functions

Arch.	Function	Signature	Description
		Basic components	
PC	pc	$\mathbb{N}$	program counter
IW	iw	$\mathbb{N}$	instruction word
		Pipeline Model	
	$full$	$\mathcal{S} \rightarrow \mathbb{B}$	stage active
	$stall$	$\mathcal{S} \rightarrow \mathbb{B}$	stage stalled
	$cancel$	$\mathcal{S} \rightarrow \mathbb{B}$	stage is canceled
	$inject$	$\mathcal{S} \rightarrow \mathbb{B}$	inject launch instr.
	$dispatch$	$\mathcal{S} \rightarrow \mathbb{B}$	dispatch micro instr.
	$last\_stage$	$\mathcal{S} \rightarrow \mathbb{B}$	instr. leaves pipeline
		Datapath Model	
R	$current_R$	$\mathcal{D}_R$	implementation register
R	$write_R$	$\mathcal{S} \rightarrow \mathbb{B}$	stage will write
R	$dest_R$	$\mathcal{S} \rightarrow \mathcal{I}_R$	write destination
R	$data_R$	$\mathcal{S} \rightarrow \mathcal{D}_R$	write data
R	$valid_R$	$\mathcal{S} \rightarrow \mathbb{B}$	data is valid
		Interfaces	
	$ibus\_read$	$\mathbb{N} \rightarrow \mathbb{B}$	instruction fetch
IF_TA	ta_{IF}	$\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$	transaction
IF_RDATA	$rdata_{IF}$	$\mathbb{N}$	read data

G. Generating The Property Suite

In order to adapt the general processor model to the actual DUV, the user needs to specify the mapping functions described in Sections IV-C to IV-E. The user input is summarized in Table I. Besides the basic data on the pipeline, given by a set of constants, the table shows the mapping functions corresponding to the architectural components of the general processor model.

During the generation of the property suite, an architecture register $R(i)$ is replaced by an instantiation of the function $Data_R(s_{fwd}, i)$, where s_{fwd} is the forwarding target stage, which is usually the decode stage.

The generated properties prove the correctness of the instructions on the implementation level. For this, we define t_0 to be the timepoint when the respective instruction enters the pipeline and $t_i > t_{i-1}, 1 \leq i \leq n$ to be the timepoints when the instruction is allowed to proceed from stage i (see also Fig. 4–6). The properties have an implication structure

$A \Rightarrow C$. Whenever the assumptions A evaluate to true, the prove part C must hold as well. In the following, we give an overview of the templates for instruction execution without exceptions. There are similar templates for exceptions; for more details, see [7]. Note that there are two templates for conditional branches depending on whether the branch is taken or not. In this way, branch prediction can easily be modeled. The assume part for an instruction m basically consists of the following assumptions:

- The instruction enters the pipeline at t_0.
- In decode, instruction m is triggered.
- The instruction proceeds from stage i at t_i, $i \leq 1 \leq n$.
- The instruction is not canceled by preceding instructions and not replaced by an exception call.

For each instruction, mainly the following will be proved:

- The instruction is fetched from the instruction memory
- The program counter is updated correctly
- The full stages are correctly tagged by the $full$ function
- No $cancel$ is generated (except for jump instructions)
- All read registers are valid
- The registers will be updated (or remain stable) corresponding to the ISA; this includes the verification of correct forwarding
- The correct transactions will take place on the interfaces

H. Completeness

The pipeline model is built such that the final property suite in combination with the consistency assertions is complete by construction, if some rules are respected for the definition of the mapping functions. For a proof for the basic pipeline model, see [7].

However, the completeness of a concrete generated property suite additionally depends on the proper definition of some of the functions. If, for example, the user defined the function for a read transaction by simply returning true, it is obvious that the interface signals are not checked at all for read transactions and there is a gap in the verification. In summary, the generation ensures that all possible scenarios are covered with properties, but not that all transactions verify all outputs. However, the automatic gap detection of OneSpin 360 can be used to close these gaps as well.

V. Application

The above method has been implemented as a front-end for OneSpin 360 MV; we call it FISACo (Formal Instruction Set Architecture Compiler). It takes an architecture description and automatically generates the instruction properties and the consistency assertions in a form readable for 360MV. The mapping information needs to be supplied in a temporal logic format. The processor model formed by both the architecture description and the mapping information can then be verified and debugged using 360MV.

In the following we will describe the application of the proposed method on an industrial processor design. We successfully verified a control processor that is used in automotive applications, the *Peripheral Control Processor* (PCP) by Infineon Technologies. First, the basic data of the PCP will be given, followed by a presentation of the verification results. Besides, during its development, the method has been applied for the complete verification of smaller processor designs from the opencores site (`www.opencores.org`). Details cannot be given here due to page limitation.

A. PCP Processor

The PCP processor is a control processor that is part of automotive systems. Its main purpose is the monitoring of peripheral components in order to release the central CPU [11]. Therefore, a great share of the instruction set is dedicated to data transfer and bit operations, which are frequently used in typical control applications. The PCP is connected to a data memory and a pipelined FPI bus (*Flexible Peripheral Interface*). As the bus operations require complex protocol transactions, 35% of the instruction set are multicycle instructions. In total, the PCP has 66 instructions, divided into arithmetic/logic instructions, jump and control, memory instructions, bus instructions and complex math instructions.

The processor is implemented as a four stage pipeline. The register file contains 8 registers of 32 bit, where one register is a special purpose register containing various status flags and the program counter. The whole RTL implementation adds up to about 17.000 lines of VHDL code, accompanied by a detailed informal specification. Regarding the complexity of the design and the quality of the source code and documentation, the time for the formal verification was estimated with 8 to 10 person months, needed to manually write a complete property suite using OneSpin 360 MV.

B. Results

The PCP has been verified using the presented approach. The informal specification was ported to an architecture description. Most of the manual effort was spent for the definition and refinement of the pipeline and datapath model, given by the mapping functions explained in Sect. IV-C to IV-E. Using our approach, the instruction set of the PCP could be successfully verified except for two highly complex bus instructions involving nested loops and excluding three complex math instructions. For these instructions, the control mechanisms of the PCP did not match our general pipeline model. It does not seem useful to extend the model for these cases, as they are very specific to the PCP implementation. Instead, having found a good representation of most of the functionality based on our processor model, the defined functions can be reused for further manual verification. This has also been done for some additional functionality beyond the ISA, like loading and storing full register contexts. The overall verification of the PCP with our methodology took about 5 person months. Thus, we could achieve an estimated productivity gain of 100%.

The verification has been carried out on 2.2 GHz workstation with 16 GB memory. Details on the proof times and the used memory can be found in Table II. As can be seen, the generated consistency assertions could be proved quickly. Most of the time was spent for the verification of arithmetic and bus instructions. The latter ones are mostly multicycle instructions and thus the design needs to be unrolled for up to

TABLE II
VERIFICATION RESULTS

Category	Properties	Total time	Avg. time	Memory
assertions	34	600 s	17.6 s	937 MB
arithmetic	15	18788 s	1252.5 s	2500 MB
logic/bit	18	3368 s	187.1 s	2500 MB
jump	7	220 s	31.4 s	2500 MB
memory	16	2135 s	133.4 s	2500 MB
bus	10	3214 s	321.4 s	2500 MB
other	3	147 s	49.1 s	1763 MB
total	103	7:54 h		

26 cycles. Note that the two most difficult instructions make up 3 hours and 48 minutes or 48% of the total runtime.

VI. Conclusions

We have presented an approach for the automatic generation of a complete property suite from an architecture description of a processor. There is a clear distinction between the architecture model and the mapping information connecting architecture to RTL implementation. The architecture model can be easily derived from an informal specification.

The mapping from the specification to the implementation is based on a general pipeline model that reflects the designer's intent in implementing a correct pipelined processor. A set of consistency assertions is automatically generated to check the correctness of the model and helps the user in finding a suitable mapping. When the mapping and the architecture description are finished, the generated property suite forms a model of the design, i.e. the verification is exhaustive.

The practicability of the approach has been demonstrated on an industrial processor design, a control processor from the automotive domain. With the presented methodology, the estimated verification productivity could be doubled.

In the future, we want to integrate this approach with our automatic generation of efficient instructions set simulators (ISS) [21]. This allows to generate both a complete property suite and an efficient ISS from a common architecture description, ensuring that the generated ISS complies to the verified RTL code. A complementary extension would be the use of existing ADL like *LISA*, facilitating the integration of formal methods into the tool chain for processor design.

References

[1] M. Aagaard. A hazards-based correctness statement for pipelined circuits. In *CHARME*, pages 66–80, 2003.

[2] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul. Putting it all together—formal verification of the VAMP. *STTT Journal*, 8(4–5):411–430, Aug. 2006.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[4] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards equivalence checking between TLM and RTL models. In *Int'l Conf. on Formal Methods and Models for Codesign*, pages 113–122, 2007.

[5] J. Bormann. *Vollstaendige funktionale Verifikation*. PhD thesis, University of Kaiserslautern, 2009.

[6] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno. Complete formal verification of TriCore2 and other processors. In *Design and Verification Conference (DVCon)*, 2007.

[7] J. Bormann, S. Beyer, and S. Skalberg. Equivalence verification between transaction level models and RTL at the example of processors, 2008. European Patent Application, publication number EP1933245.

[8] J. Bormann and H. Busch. Method for determining the quality of a set of properties, applicable for the verification and specification of circuits, 2007. European Patent Application, publication number EP1764715.

[9] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on CAD*, 23(12):1625–1639, 2004.

[10] C. Braunstein and E. Encrenaz. Formalizing the incremental design and verification process of a pipelined protocol converter. In *IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, pages 103–109, 2006.

[11] S. Brewerton. Dual core processor solutions for IEC61508 SIL3 vehicle safety systems. In *Embedded World Conference*, 2007.

[12] R. Brinkmann, P. Johansson, and K. Winkelmann. *Advanced Formal Verification*, chapter Application of Property Checking and Underlying Techniques, pages 125–166. Kluwer Academic Publishers, 2004.

[13] R. Bryant, S. German, and M. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):93–134, 2001.

[14] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 68–80, 1994.

[15] W. Büttner. Complex hardware modules can now be made free of functional errors without sacrificing productivity. In *Proc. of Int'l Conf. on Abstract State Machines, B and Z*, LNCS, pages 1–3. Springer, 2008.

[16] A. Chattopadhyay, A. Sinha, D. Zhang, R. Leupers, G. Ascheid, and H. Meyr. Integrated verification approach during ADL-driven processor design. In *IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, pages 110–118, 2006.

[17] G. Fey and R. Drechsler. Design understanding by automatic property generation. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 274–281, 2004.

[18] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 521–537, 2000.

[19] D. Kroening and W. J. Paul. Automated pipeline design. In *Proc. of the 38th Conference on Design Automation*, pages 810–815. ACM, 2001.

[20] U. Kühne. *Advanced automation in formal verification of processors*. PhD thesis, University of Bremen, 2009.

[21] U. Kühne, S. Beyer, and C. Pichler. Generating an efficient instruction set simulator from a complete property suite. In *Proc. of the IEEE Int'l Symposium on Rapid System Prototyping*, 2009.

[22] P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *Proc. of the Int'l Conf. on Computer Aided Design*, pages 863–870, 2005.

[23] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on CAD*, 27(11):2068–2082, Nov. 2008.

[24] OneSpin Solutions GmbH, Munich, Germany. *OneSpin Verification Solutions*. http://www.onespin-solutions.com, 2009.

[25] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. In *Design, Automation and Test in Europe*, 2008.

[26] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 135–146, 1998.

[27] E. Schnarr, M. Hill, and J. Larus. Facile: a language and compiler for high-performance processor simulators. In *Proc. of the Conf. on Programming language design and implementation*, pages 321–331, 2001.

[28] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, pages 127–144. Springer, 2000.

[29] J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *Proc. of the Int'l Conf. on Computer Aided Verification*, pages 98–109, 1998.

[30] M. Velev. Formal verification of pipelined microprocessors with delayed branches. In *Int'l Symp. on Quality Electronic Design*, page 4pp., 2006.

[31] M. Wedler, D. Stoffel, and W. Kunz. Exploiting state encoding for invariant generation in induction-based property checking. In *Asia and South Pacific Design Automation Conference*, pages 424–429, 2004.