Complete Formal Verification of TriCore2 and Other Processors

Jörg Bormann, Sven Beyer, Adriana Maggiore,
Michael Siegel, Sebastian Skalberg
OneSpin Solutions GmbH
Munich, Germany
<firstname>.<lastname>@onespin-solutions.com

Tim Blackmore,
Fabio Bruno
Infineon Technologies
Bristol, UK
<firstname>.<lastname>@infineon.com

Abstract

This paper describes an innovative and powerful methodology for the complete formal verification of modules and intellectual property (IP), and its application to the verification of processor IP. Unlike other formal approaches, the methodology is a self-contained approach to hardware verification, independent of simulation. The methodology eliminates all gaps in the verification plan and in the property set. It thus ensures that the IP is free of functional errors – the highest possible verification quality. Its underlying technology has been field-proven on hundreds of modules and IP, two of which are described, including the TriCore2 processor, Infineon's next generation high-end processor for embedded and safety-critical applications.

I. Introduction

The objective of verification is to ensure that implemented behavior and specified behavior are the same. Intended behavior that is not explicitly captured in the written specification will be referred to as "implicitly specified behavior" or "implicit specification". In order to eliminate all deviations of the implemented from the specified behavior, verification methodologies must consider every possible input scenario to the design-under-verification (DUV) and verify that every possible output signal has its intended, specified value at every point in time.

Established verification approaches, however, typically do not identify all functional errors in a design. Errors that remain undetected and make it into the final chip generally fall into one of three types:

- Unstimulated error: an error where the input stimuli used for verification fail to trigger the error and thus prevent its observation.
- Overlooked error: an error that is stimulated, but where there is no property, monitor or assertion to observe and flag the erroneous behavior.
- Falsely accepted error: an error where erroneous behavior is not detected because both the RTL implementation and the properties, monitors or assertions deviate in the same way from the specified behavior, thus masking the error.

Simulation-based verification and traditional formal verification have different strengths and weakness in identifying such erroneous behavior.

A. *Simulation-based Verification*

Simulation-based verification approaches suffer from all of these error escape types.

Simulation fails to stimulate errors because it cannot deploy the multitudinous stimuli necessary to exhaustively verify the IP in the project time available. Simulation coverage metrics cannot relieve this situation – they can only assist in the allocation of restricted verification resources to measure progress and to incrementally increase the quality of the verification.

Overlooked errors are handled by verification planning and the subsequent derivation of suitable monitors and assertions. Verification tasks are identified by (i) examination of the specification and the architecture; (ii) relating common design patterns to appropriate assertions [3]; or (iii) asking the designers to note particularly important relations between given signals. The completeness of the resulting verification tasks is typically compromised by the inability to devise monitors that inspect all output signals constantly for all possible errors that might occur, and by human error. Consequently, it is essential to update the plan throughout verification in order to capture new insights into unmet verification needs [1].

B. *Traditional Formal Verification*

Generally speaking, formal verification has similar problems. Single properties are exhaustively proven with respect to all possible input scenarios. However, the properties typically have an implicative structure. When a given input pattern occurs (for instance, a write request is received by a bus arbiter), statements about the DUV state and output behavior are made, (for instance, an acknowledgement is transmitted). Thus a single property typically verifies only a fraction of the possible input scenarios and their associated output behaviors. This leads immediately to the central questions in formal verification: "Have I written enough properties, or are there gaps in the property set? Is every possible input scenario inspected and its effect on the states and outputs verified by at least one property?"

This situation is exacerbated by the fact that most assertions and properties can be proven only under specific conditions that enforce realistic behavior of the DUV. These conditions must be identified by the verification engineer, who does not always possess the design functionality knowledge necessary to correctly capture them. Moreover, traditional formal verification typically detects corner case problems only if the verification engineer anticipates them – a major challenge in complex designs. In other words, formal verification approaches have not been able to produce the gap-free property set that ensures an error-free design.

The result is that – until now – formal verification has not enabled engineers to achieve *complete* verification. Formal verification has been operating below its actual potential [5]. In reality, it has often been downgraded to verify only some aspects of a design in order to reduce simulation run time.

However, formal verification can efficiently achieve the essential completeness objectives when it deploys systematic *completeness analysis* to identify gaps in a property set.

C. Formal Verification With Completeness Analysis

The approach taken with this new methodology is different from traditional property checking, in that it investigates the quality of a set of properties. An automatic completeness analysis formally checks whether a given set of properties is stringent enough to determine a unique value for every output of a module at every point of time, and to do so for every possible input stimulus. It automatically identifies and highlights input scenarios and respective output behavior that are not yet verified by any property, and insists upon its resolution. This ensures that all functionality that contributes to the input/output behavior of the design is thoroughly checked by at least one property. Consequently, it eliminates gaps in the property set, i.e. scenarios where the property set misses erroneous implemented behavior.

Using this methodology, a set of properties must pass both verification flows in Fig. 1: in the left, standard property checking flow, the design is formally verified against every property, while the right flow is used to incrementally identify and close all gaps in the property set until completeness is reached. The right flow is unique to the presented formal verification approach and opens the door to a previously unachievable level of functional verification quality and productivity.

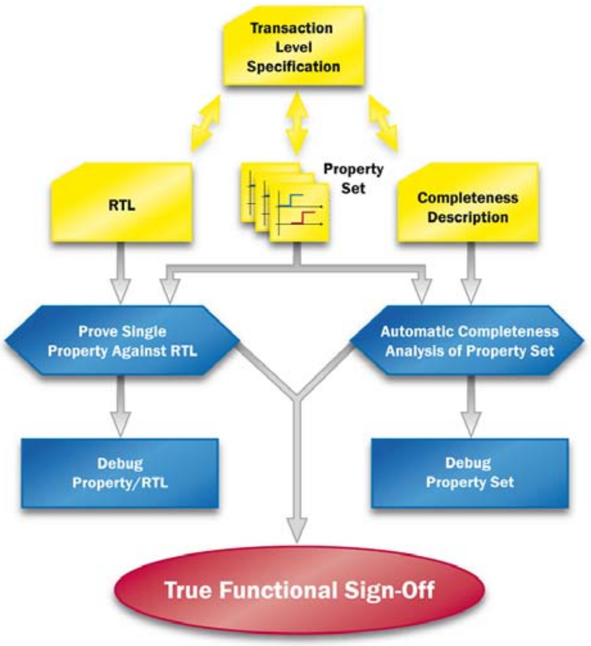
Transaction Level Specification
RTL
Property Set
Completeness Description
Prove Single Property Against RTL
Automatic Completeness Analysis of Property Set
Debug Property/RTL
Debug Property Set
True Functional Sign-Off

Fig. 1: OneSpin 360 MV Verification Flow

The combination of these two flows eliminates – in the terminology of the error classification previously defined – unstimulated errors and overlooked errors. Furthermore, a tailor-made completeness methodology complements the formal verification to minimize the risk of falsely accepting errors. If a property set, developed according to this methodology, passes on both flows, the verification is complete – the only safe criterion to terminate verification. At this point, a true functional sign-off for the DUV has been achieved.

Formal verification with completeness analysis is a self sufficient verification solution that:

- Is independent of simulation.
- Detects all functional errors.
- Implements a precise and objective verification termination criterion, independent of human and heuristic factors.
- Improves specification quality by systematically identifying omissions and errors in the specification.
- Delivers a high verification productivity of 2 k to 4 k lines of verified RTL code per engineer-month [6].
- Is complemented by a lean methodology that affords a high level of design visibility by relating

the transaction view of a design to its implementation.

- Integrates smoothly into existing design and verification flows.

The completeness analysis shifts the focus from a single property to the quality of an entire property set. When completeness analysis identifies a gap in the property set, and the missing information is not found in the specification, completeness analysis has identified a specification gap and provides the opportunity to close it. Thus the methodology also systematically improves the quality of the specification.

D. Structure of the Paper

Completeness analysis is described in section II. It is used in conjunction with a methodology that guides property development. Once adopted, the verification methodology can be employed to verify a large variety of designs. The base of the methodology clarifies how a design is verified with a set of properties that all belong to a single completeness plan. Section III describes this base methodology with specific reference to processors, and provides an application example using a small processor.

In order to handle larger designs – up to several hundred thousand lines of RTL code - they must first be partitioned into multiple sub-designs that can be verified with one completeness plan. This provides a clear understanding of the input/output behavior with a precision that enables the combination of the results for the sub-designs by manual reasoning. This step is described in section IV, using the TriCore2 as an example.

II. Completeness analysis in depth

Single properties are used to describe the effect of single transactions, while a sequence of transactions is captured by a chain of properties. The automatic 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.

Completeness analysis uses properties of an implicative style, with descriptions of the input scenario for which the property is developed and the associated expected behavior. The input scenario consists of a state and input descriptions. The expected behavior consists of output and state expressions.

An example of the completeness analysis approach is shown in Fig. 2. It begins with at a reset state. For every possible input scenario, it is first checked that a chain of properties can be built such that the state descriptions of adjacent properties are consistent. Secondly, the properties along these chains are analyzed to ensure that they determine a unique value for every output at every point of time.

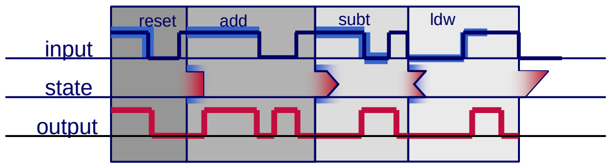

Fig. 2: Chains examined by completeness analysis

Fig. 3 shows examples of gaps that completeness analysis identifies. In the first example, an input scenario is identified that is only partially covered by a property chain. None of the existing properties matches at the end of the chain, so a portion of the input scenario has not been covered yet. This gap could allow errors to escape detection.

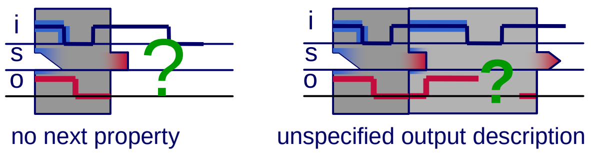

Fig. 3: Example gaps identified by completeness analysis

The second example is that of an input scenario for which a full property chain exists, but where the properties do not determine a unique value for a given output(s) at some given point in time. In this case, the gap could allow erroneous output behavior to be overlooked and thus escape detection.

Despite these gaps, the design may behave as intended. However, the completeness analysis reveals that the property set is not gap-free and thus does not fully capture the behavior of the DUV. Consequently, the property set cannot ensure the absence of unintended behavior.

The completeness plan that is used by the completeness analysis consists of:

- A property graph that specifies which properties can succeed each other during the process of building chains of properties.
- A list of signals that should be treated as inputs
- A list of determination requirements that allows the engineer to specify which output signals or expressions must be determined by the property set, and when.
- A reset property.

The property graph is a high level view that captures the intentions of the verification plan. If the graph does not comply with the properties themselves, completeness analysis flags the discrepancy, enabling a focused diagnosis.

III. Operation Properties

The presented verification approach is used in combination with a methodology how to develop properties for a given design. A single, so-called *operation property* expresses and captures a single design transaction, which is a transition between abstract, high level design states. This transaction view of the design is an abstracted view that, being free of implementation detail, can be easily compared against the natural language specification.

Properties have an implicative structure consisting of an "assume" and a "prove" part. The "assume" part of an operation property describes the abstract, high-level state where the transition starts and the condition under which the corresponding operation is executed. The "prove" part describes the expected output resulting from the operation and the abstract state where the respective transition ends.

The transaction view of the operation properties is tied to a specific implementation via a set of mapping functions that map transaction-level entities – such as abstract, high-level states and conditions – to implementation-level entities of the DUV. The same mapping functions are used in all operation properties; they are easily readable and far more compact then the RTL code itself.

For instance, the transaction view of a property can express the expected register-level behavior of an ADD instruction in a processor design in a few lines. The mapping function, which describes, for instance, all data forwarding, consists of a few hundred lines – resulting in the detailed signal view of the property. The underlying RTL code comprises several thousand lines. The transaction view thus affords a high level functional view that considerably simplifies verification.

A. *Operation Properties for Processors*

Smaller, single pipelined processors can be examined with a complete set of operation properties without further subdivision. For such processors, the transaction view of the properties primarily captures the architecture description, which is used to define the contents of the programmer's manual with its instruction descriptions. The obvious classes of operations are therefore related to the instructions: every operation class describes the execution of one instruction. There is only one high-level state from which each instruction execution starts and to which each returns. The transaction view of the related property then expresses that:

- A read transaction is made on the instruction bus for which an abstract program counter provides the address.
- The instruction read with this transaction is provided with data from some abstract register file, a program status word, etc.
- This data is correctly processed according to the architecture specification involving, if necessary, read or write transactions on the data bus.
- The result correctly updates the abstract registers.
- The abstract program counter is updated correctly.

Similar properties are developed for interrupts. This transaction view captures only the architecture description. It is therefore independent of – and applicable to – all architecture implementations.

To transform these generic transaction view properties into signal view properties, all abstract values mentioned above are mapped to appropriate expressions about implementation signals.

This mapping, which is used by all operation properties, typically involves controllers for multi-cycle instructions, etc. The proof of the instruction therefore ensures that these controllers always properly return to their idle states, and that the execution of an interrupt does not cause any undesired side-effect on the implementation state that would disturb the instruction execution later on.

The replacement for the abstract register file – resulting from the foregoing mapping – is referred to as a virtual register file. It is a function that tests the signals in the implementation of successive pipeline stages in order to determine whether they correctly store results for a given register file address. It returns the first result that it finds, as well as the related value of the register file of the implementation when there is no such value in the pipeline. Thus, the virtual register file describes how forwarding provides an instruction at the beginning of a processor pipeline with appropriate data. This transforms the verification of forwarding into a by-product of the proof of the related property.

The replacements for the abstract reads and writes on the data and instruction busses describe the signal behavior for the related transactions. The replacement for the abstract interrupt input defines how interrupts are merged into the instruction stream.

In general, the expressions that replace the abstract values in the transaction view involve time shifts to account for the pipeline structure of the implementation. The implementation program counter provides the address of a store instruction several clock cycles before the output signals to the data bus initiate the store transaction. In effect, the signal view property accompanies the instruction while it moves down the pipeline and relates the proof goals about signals in every pipeline stage to those points in time when the stage participates in the execution of the instruction.

The resulting signal view of the property typically makes no assumptions about other instructions before or after the one being examined. Therefore, the single formal proof of this property against the implementation shows that the execution of the current instruction is not impacted by any precedent instruction. Thus, this proof typically replaces many separate verification tasks otherwise necessitated by the consecutive execution of all possible pairs of instructions.

Completeness analysis is employed to identify and subsequently close all gaps in the set of properties. If the completeness of the resulting set of properties is confirmed, it is ensured that every execution trace of the processor can be captured by a sequence of signal view properties (cf. Fig. 4). Their related transaction view shows that this executes a program according to the specification. Therefore, the proof of the properties against the implementation, together with completeness analysis of the property set, ensures functional equivalence between the implemented processor and its architecture.

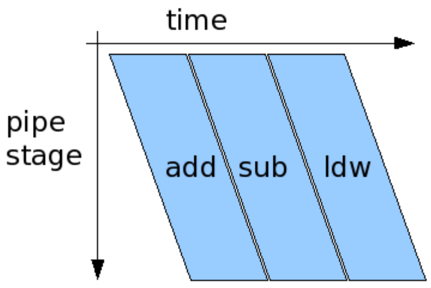

Fig. 4: Sequence of instruction specific properties

One gap often identified by completeness analysis in relation to this approach is a so-called bubble. This type of gap describes stages that fall empty because they were allowed to pass an instruction to the next stage while the previous stage stalled. The related properties must prove that such empty stages do not modify the virtual state and do not access the memory.

B. *Application Results*

This formal verification methodology with completeness analysis was applied to a 32 bit Infineon protocol processor with 40 instructions. The functional space of this processor is quite large, due to its configurability and sophisticated multithreading support with four contexts. Because of this large functional space, it seemed most expedient to apply functional verification using the above methodology. Simulation was used where it was helpful during the design process, but the project relied solely upon formal verification for functional verification.

Despite the intricacy of the processor, only 40 operation properties were required to completely describe its functionality. After 4 engineer-months, the property set was proven against the implementation and completeness analysis confirmed the absence of gaps. This terminated the verification. The extremely high quality delivered by this verification approach was demonstrated by the fact that in none of the several system-on-chip (SoC) projects that integrated this internal IP were further errors detected during either the ensuing system verification activities or in field operation.

All errors were removed during the verification phase. Intricate corner cases, such as the interaction of delay slots and context switches, were systematically and efficiently explored, and identified related problems were eliminated with a partial, targeted redesign.

A simulation-based verification of a previous processor of comparable size required nearly twice the effort and did not deliver as high a quality.

C. *Quality Assessment*

The complete formal verification of the processor produced a strong correctness statement. As discussed earlier, the combination of the property checker and completeness analysis detects all unstimulated and overlooked errors.

However, the question that remains is to what extent the verification can avoid falsely accepting errors. To falsely accept errors, the RTL implementation and the properties must misinterpret the specified behavior in the same way. This risk is minimized by the approach described above. The architecture specification is translated almost verbatim into the transaction view of properties. The simplicity and conciseness of the properties – especially in comparison with exceedingly large testbenches – make them easily reviewable against the specification. Furthermore, independent modeling of the implemented behavior by compact properties reduces the probability of repeated modeling errors. All other user inputs about internal circuitry, such as the virtual register file, the replacements for the abstract program counter, or the abstract state, cannot lead to falsely accepted errors, because they serve as induction hypotheses along the chains of properties under examination by completeness analysis. This means that all these user inputs are checked by one property before they are assumed by the next property. If the user input does not reflect the design logic, the property cannot be checked against the design, leading to verification failure. Of course, there might be different user inputs with which the verification would succeed, but this scenario cannot mask an error.

This is significantly different from simulation-based verification, in which the critical internal parts of the design (such as forwarding) and the specification of checkers to secure their proper operation are crucial. Such checkers are specified without the ability to ensure that they indeed fully capture the design's proper operation. This approach thus runs the risk of falsely accepting incorrect behavior.

IV. TRICORE2 VERIFICATION

TriCore2 is Infineon's second generation 32-bit processor. It is a RISC processor with added DSP and microcontroller capabilities, targeted at high-end embedded applications, including safety-critical applications. It is superscalar, multi-threaded and has fast context-switching capabilities. It has two sets of 16 general purpose registers (GPR) – one for data and one for addresses.

A. Planning for Complete Formal Module Verification

The formal verification of a complex processor such as TriCore2 is necessarily a complex task. The effective execution of such a large formal verification project requires thorough planning, so that the verification can be carried out by a team of engineers working in parallel, leveraging the completeness methodology. This section describes the verification planning of large projects, using TriCore2 as an example.

In the planning phase, one or more top level verification goals are successively decomposed as shown in figure 5.

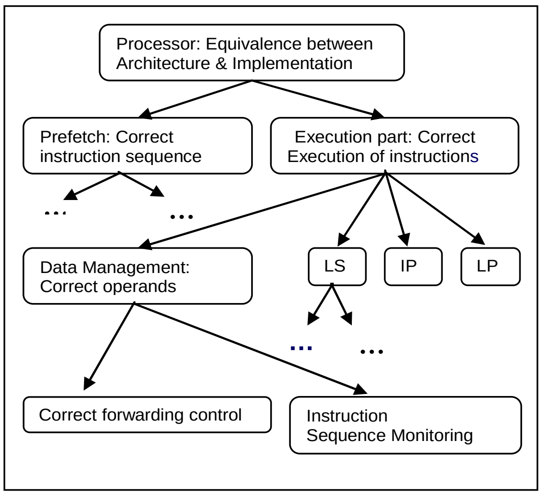
Processor: Equivalence between Architecture & Implementation
Prefetch: Correct instruction sequence
Execution part: Correct Execution of instructions
...
...
Data Management: Correct operands
LS
IP
LP
...
...
Correct forwarding control
Instruction Sequence Monitoring

Fig. 5: Verification goal decomposition

This leads to the definition of sets of parallel and serial tasks. Some tasks – serial tasks – can be more easily executed if other tasks have been previously executed, because the earlier tasks clarify issues required by later tasks. Such dependencies must be identified. Other tasks – parallel tasks – are independent of each other, and can be executed by different team members with little co-ordination. Effort estimates are then calculated for each task. Finally, a graph of tasks and their dependencies is created. Using this graph, the parallelism inherent in the verification project is determined, so that the tasks can be efficiently distributed among a team of verification engineers. This effort estimation determines the project schedule. Project progress is reported on the base of the tasks and their schedule.

An important observation from the verification plans created according to this approach is that very often some relatively small number of top level verification goals suffices to start the decomposition of figure 5. For TriCore2, the overall objective is the same as for the protocol processor previously discussed, namely, the equivalence between the architecture description and implementation.

The decomposition of the top level objective proceeds in stages. In the first stage, the functions of fetching and execution are identified, and the top level requirement partitioned into the proofs i) that the instructions are fetched from the program memory and sent to the pipelines in the correct order and ii) that the instructions received by the pipelines are executed according to the specification.

The verification of both fetching and execution functions is then further decomposed. The execution of a given instruction is proven by i) the provision of the operands and ii) the execution within the target pipeline to produce a modification of the architectural state, calculation of the result of an arithmetic instruction, loading of data from memory, etc.

The correct provision of an instruction with operands is decomposed into the proofs that i) the instructions moving in the pipelines correctly signal when they provide a result and when the result is available (forwarding control); ii) the instructions are stalled when their source operands are not available (data dependency stalling), and iii) the implementation keeps track of the correct sequence of the instructions (instruction sequence monitoring) − a non-trivial task because instructions are allowed to move at different speeds in the pipelines.

Assuming that an instruction receives the correct source operands, the correct execution in each pipeline is verified separately for each of the three Tricore2 pipelines, the Load/Store pipeline (LS), the Integer Pipeline (IP) and the Loop Pipeline (LP). The verification of the LS pipeline is further decomposed for classes of instructions, for instance, for those that access the memory management unit (MMU).

The last step in the decomposition process partitions some proof goals into manageable tasks by the introduction of temporary restrictions. For example, the verification of the instruction sequence monitoring in the Tricore2 was first defined on the assumption that there was no cancellation; a second phase allowed for cancellations due to wrong speculation, but not due to exceptions, and the final phases verified the functionality without restrictions. This stepwise removal of restrictions enables the verification engineer to gradually build knowledge of the full design functionality and to finer track verification progress.

The final project plan of TriCore2 allowed three verification engineers to work on the project in parallel, condensing 8 engineer-years of effort into less than 3 years real time. Most tasks are executed according to the methodology for designs that can be verified with one verification plan. This effort amounted to around 90% of the total verification effort. The results were then analyzed by informal reasoning to deduce the proof goal for the next level of decomposition. This combination of results followed the reverse order of the decomposition, all the way up to the top level verification objective of equivalence between the instruction set architecture (ISA) and the RTL implementation.

B. *Quality Considerations*

This decomposition procedure highlights intermediate proof goals, similar to the verification objectives of verification plans for simulation. However, for complete formal verification, these proof goals are typically expressed with some level of freedom to comprehend the actual results in the course of verification. This level of freedom does not impact the final verification quality because, ultimately, these intermediate goals must be proven if they are used in other proof tasks. Therefore, intermediate proof goals have no impact on the quality that the verification will finally produce. This is different from verification plans for simulation, where proof goals about internal behavior are identified to provide better visibility into the design and thus to increase the number of detected errors.

Verification planning for complete formal verification has a major impact on schedule reliability. If the decomposition into tasks does not reflect the architecture of the design, the combination of the partitioned tasks may fail, and the project plan would have to be re-worked.

The manual reasoning involved in combining the results of verification tasks starts from precise descriptions. It is therefore possible to exploit the rigor of mathematical proof. To minimize the impact of human fallibility during this step, intermediate results can be packaged into macros of the property language which are proven in one task and assumed in the other.

C. *Example Verification Task*

As an example, we consider the data management tasks for the integer pipeline. This involves proving properties for the correct behavior of forwarding control signals as well as combining the related results with the results of other tasks to ensure the proper delivery of operands.

The inputs to the forwarding control functionality are the opcode, stall and cancel signals. The forwarding management signals are for each pipeline stage:

- VPTR: set if the instruction produces a result.
- VRES: set if the result of the instruction is valid in the pipeline stage.
- VPSW: set if the instruction modifies the status word.
- Signals carrying result and destination register address.

The IP instructions are grouped into classes with the same expected behavior of the forwarding management signals. For example, the class *early_vres* groups the instructions that produce a result in the first execution stage ex1. A property is written for each class of instructions. The property describes the expected behavior of the VPTR, VRES, VPSW result and the destination address signals from the point in time at which the instruction is ready to leave the decode stage, dec, to the time it leaves the pipeline. For example, the early_vres property describes how:

- $VPTR_{dec}$ and $VPSW_{dec}$ are set when the instruction is in the decode stage, while $VRES_{dec}$ is not set.
- $VPTR_{ex1}$, $VPSW_{ex1}$, $VRES_{ex1}$ are set when the instruction moves into the first execution stage ex1 and remain set until the ex1 stall signal is removed.
- $VPTR_{ex2}$, $VPSW_{ex2}$, $VRES_{ex2}$, $VPTR_{wb}$, $VPSW_{wb}$, $VRES_{wb}$ are set when the instruction moves into the write-back stage wb.
- The destination register address propagates correctly through all stages.
- The result signal propagates from the point of time onwards where the $VRES_{ex1}$ signal becomes true.

Results of the forwarding control tasks (as sketched above) were combined with the results of the data dependency stalling and instruction sequence monitoring tasks to show that the instructions are provided with the correct operands. The instruction sequence monitoring task takes the application of new instructions in the three pipelines as inputs and proves the correct setting of the instruction tags, so that the sequencing of the instructions in the pipeline can be established. The data dependency stalling task takes the attributes of the verification operands as inputs, proven by the forwarding control task, and the instruction tags, proven by the instruction sequence monitoring, and proves that an instruction stalls in the decode stage until its source operands are ready for forwarding.

D. *Example Error*

During a similar verification for the load store pipeline, an error was found that is one example of the type of problems that complete module verification detects.

The error occurs with an instruction sequence similar to the following:

(1) LOAD D2

(2) ADD D0=D2+0

(3) ADD D6=D4+0

(4) ADD D8=D2+0

(5) STORE D8

Instruction (5) must be misaligned, that is, taking two accesses to complete. Further on, an external write to one of the core special function registers needs to be executed during the second access of instruction (5). Then the old data in register D8 (i.e. before instruction (4) writes to D8) is stored, contrary to the expected behavior, where the result of (4) is to be stored. This clearly is an obscure corner case error highly unlikely to be triggered in simulation – which is not to say that it is highly unlikely to occur in real life. Since it depends upon a particular code sequence, it has a software workaround. Therefore, it is not classified as critical, but it is crucial to find these types of error during verification to avoid failure in later safety-critical application. Its correction is supported by formal verification, because the set of instruction sequences that make the error occur could be precisely identified.

E. Error Statistics

The verification of TriCore2 combines formal and simulation-based approaches. The simulation-based dynamic verification of TriCore2 is detailed in [2]. Essentially, it consists of a large suite of assembly tests, some hand-written, some Perl-generated and some generated using a Specman Instruction Stream Generator (ISG). The ISG is capable of generating both highly directed and highly random tests. The tests can be run on a configurable, directed testbench, or on a constrained-random testbench. Configuration options include where the test is placed in memory, whether the MMU or external coprocessor is enabled, etc. The directed testbench is designed to generate external bus traffic, interrupts and idle requests at critical times. The random testbench generates these stimuli randomly, exploring unforeseen problem areas.

Checking is carried out on several levels. Many tests are self-checking, jumping to fail for unexpected results. Beyond this, an instruction-by-instruction comparison of the traces produced by the design is compared with that produced by the golden model. A comparison of the memory contents is made at the end of the test and OVL assertions are embedded in the design. Sign-off criteria included both structural and functional coverage targets (e.g. 100% statement and branch coverage, 100% of defined functional coverage hit). This involved considerable effort in terms of engineering time, license use, and use of computer resources.

Errors found during TriCore2 verification were tracked and information about the methodology that detected the error and its severity was recorded. Errors were not tracked until the directed test suite (of around 10,000 tests) had a pass rate of over 99% in the default configuration, thus ensuring a base level of design quality. After this base level of quality was achieved, the formal verification of TriCore2 found 89 errors (out of a total of 259 errors reported by all methodologies) in the design and 67 errors in the specifications.

Since the formal verification was run concurrently with the dynamic verification, some of the errors found by formal verification might also have been found by the dynamic verification. However, analysis of the nature of the errors suggests that this would apply to at most 40 of the design errors. Conversely, once the formal verification of an area of TriCore2 was complete, no other verification methodology found further errors in that area. Of the 50 design errors that would not have been found by simulation, several were classified as critical (meaning unavoidable lock-up or data corruption).

The specification errors were found during the formalization of the architecture and when the implemented instruction execution was verified against the formalized architecture description. Its number might show how precisely the architecture can be formalized. The verification discovered differences that were corrected by adapting the specification.

V. Conclusion

Existing verification techniques, both static and dynamic, can leave errors in the design undetected. The complete formal verification methodology presented in this paper combines the formal proof of individual properties, the automatic formal proof that the set of properties is a complete description of the design functionality, and the transaction-based style of the properties to prove that the design is error free. Two examples of the application of the methodology were described, the complete verification of a single-pipeline protocol processor and of the superscalar Tricore2 processor. The verification of the Tricore2 also details how complex verification tasks are managed and decomposed, showing that the methodology is applicable to large IPs and is suitable for execution by verification teams.

[1] Aycinena: Verification Test Plan: The Book : EDA Café: http://www10.edacafe.com/nbc/articles/view_weekly.php?articleid=307775

[2] Bruno, Blackmore: Verifying the Tricore2 Multithreaded Microprocessor, Design Con 2006

[3] Foster, Krolnik, Lacey: Assertion Based Design: Springer 2003

[4] Murphy, Kurshan, Albin, Wolfsthal, Geist, Sawada, Nguyen, Fix, Gorman, Edwards, Yeh, Joyner: Research Needs in Verification, April 2005 - www.src.org/fr/s200502_needs.pdf

[5] Shimizu, Gupta, Koyama, Omizo, Abdulhafiz, McConville, Swanson: Verification of the Cell Broadband Engine^{TM} Processor: DAC 2006

[6] Bormann, Blank, Winkelmann: Technical and Managerial Data About Property Checking With Complete Functional Coverage: Euro DesignCon 2005