An AMBA-ARM7 Formal Verification Platform

Kong Woei Susanto[1] and Tom Melham[2]

[1] Department of Computing Science, University of Glasgow
Glasgow, G12 8QQ, UK
susanto@dcs.gla.ac.uk

[2] Computing Laboratory , Oxford University
Oxford, OX1 3QD, UK
Tom.Melham@comlab.ox.ac.uk

Abstract. The pressure to create a working System on Chip design as early as possible leads designers to consider using a platform based design method. In this approach, designing an application is a matter of selecting from a set of standard components with compatible specifications. Subsequently, a formal verification platform can be constructed. The formal verification platform provides an environment to analysed the combined properties of the design. In this paper, we present a methodology to do formal System on Chip analysis by developing generic formal components that can be integrated in a formal verification platform. First, we develop reusable formal properties of standard components. Second, we define a generic formal platform in which components of System on Chip design can be integrated. The platform contains basic components such as a standard bus protocol and a processor. Third, we combine the properties of standard components and obtain a set of refined properties of the system. We use these properties to develop the required specifications of the remaining components.

1 Introduction

The effort to implement a single chip system from scratch is enormous and only a few companies have the needed competency in all design areas. In most cases, designers will have to use *Intellectual Property* (IP) blocks or *Virtual Components* (VCs). IP blocks are predefined, large grained logic blocks, (such as processors, memories, and peripherals) whose function has been precisely specified. They can be developed in–house or originate from external vendors. When IP blocks become widely available, the design focus will shift to reuse. Chip design is becoming much more a matter of design composition than of design creation.

These compositional or reuse based design methodologies will be the issue addressed by *System on Chip* (SoC) designers. A standard platform and application specific architectural context will play a major role in achieving a plug and play environment using reusable components. Such an *integration environment* will typically be a design platform for a specific application domain. The IP blocks will be the standard building blocks that can be easily integrated within the application domain [6,16].

The verification of SoC design is arguably the biggest challenge for designers. Complexity has made design validation the bottleneck in the completion of a design project. A new design and validation methodology is needed to address this problem [4]. This must be capable of reducing the amount of analysis and debugging that takes place in the early development stages. The new validation methodology should emphasise reusing existing validation code. This reduces the time needed in recreating the validation code. The use of an abstract representation of IP models could help to speed up the validation process.

The limits of traditional validation methods have prompted the industrial community to consider formal verification methodologies for verifying hardware system specifications and models [19]. The inclusion of formal verification will remove uncertainty, increasing confidence in the design, and reduce verification time. Advances in design and validation methodologies for system level verification will make changes to the current formal verification approach necessary.

In this paper, we present one methodology to do formal SoC verification. The core of this methodology is in the development of reusable formal properties or proofs that can be used in the development of an SoC design. The properties contain the operational conditions for the system and specify its input/output relations. These properties are used as the behavioural representations of the components. We use these properties to define the requirements for each components used in the design of an application. The methodology is based on the combination of semiformal [1] and hybrid systems [5]. Tool support for the methodology is constructed from a selection of formal tools.

The contents of the paper are as follows: In Section 2 we briefly describe the integration platform approach for system on chip technology. In Section 3 we explain our idea of a verification platform, followed by a brief description of the formal tools environment. The AMBA and ARM platform is described in Section 4. In Section 5, we describe the specification development for an Ethernet Switch system built on this platform. A summary and discussion of future work is presented in Section 6.

2 Typical SoC Architecture

Cadence, Synopsys, and Mentor Graphics, three major providers of Electronics Design Automation (EDA) tools, have proposed similar systems that support or are based on the integration platform concept. A typical integration platform [21] is presented in Figure 1. A simple general-purpose processor core is the basic component. The platform is customisable with a collection of IP blocks which can be either user-defined logic blocks or third party IP blocks. All the IP blocks communicate through busses that also communicate with the processor.

Within this environment, two kinds of busses are introduced: the processor local bus (PLB) and the on-chip peripheral bus (OPB). There will be only one PLB but there may be more than one OPB. The OPB is connected to the PLB through a module interface called the OPB bridge. The PLB arbiter controls

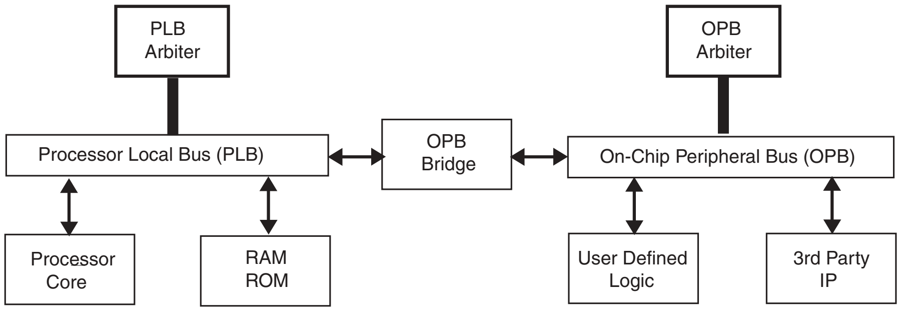

Fig. 1. A Typical Integration Platform.

the PLB communications among the processor, memory and OPB bridge. The OPB arbiter controls the OPB communications among the IP blocks.

Typically, an integration platform will come as a platform for a specific application domain. The platform will consists of components such as a target hardware and software architecture, a portfolio of VCs, and a design validation methodology. Its IP blocks are standard building blocks that can be easily integrated together into a system in the platform.

The design validation methodology determines the kinds of test-benches required for system level verification. These test-benches must be started early in the design process to avoid the possibility having a working chip but a failed system. The methodology determines which verification tools (event-driven simulation, cycle-based simulation, emulation) can be used. This information is very useful as each selected tool may have specific coding style requirements. Finally, the methodology defines a way to validate the system with an application running on it.

3 A Verification Platform Approach to SoC

In parallel with the system integration platform described above, we suggest that a *formal verification platform* is constructed. A formal verification platform is a standardised platform where a verification engineer can easily integrate various formal models in a single environment and perform formal validation of the system [26]. In this, each of the building blocks is represented as a formal specification model. There is a model for the processor core, for the bus and its protocol, and for all IP blocks available. Then the different models are integrated as a single system description in the verification platform.

Similar to validation, which commonly uses simulation, the formal verification platform may apply a variety of verification techniques. For example, a processor core formal model can be verified by symbolic simulation or by formal proof [10]. A bus protocol formal model is normally verified using a property checker [23]. The verification platform needs to accommodate all these various verification techniques.

There are two approaches to defining the formal models for these verification methods. The first is to define them all in a single specification language that has a complete set of verification techniques. HOL [8,9], PVS [24], ACL2 [14,15], and Forte [12] are the examples of this kind of verification environment. Another approach is to use a mixture of available tools, PROSPER [7] being one notable example of an environment designed for this. This approach enables the verification platform to use the most appropriate tools without compromising performance. But it has the drawback that a system might be formally modelled in more than one specification language. It also raises the issue of integrating the different tools used so that they can communicate. A logical connection among the tools is required in which the formal models can be integrated as a single system, using a kind of glue logic to connect them.

Our work uses the second approach, a mixed tools environment. We construct *a verification environment* which has the capabilities of various formal verification technologies, such as a theorem prover, a symbolic simulator, and a model checker. The verification environment combines the HOL98 theorem prover, the ACL2 theorem prover and the SMV model checker [17]. HOL98 is the centre of the tools environment. ACL2 and SMV are connected to HOL98 through a layer of interfaces. Through these interfaces, users can send commands from HOL98 to instruct ACL2 and SMV to perform formal proof. HOL98 also accepts proved theorems and properties from ACL2 and SMV as theorems in its own logic. Detailed descriptions of the system are given in the reminder of this section.

3.1 Theorem Prover

The theorem prover is the central tool to perform an integrated system level verification. Its command language is treated as the implementation language for interfacing various formal tools in the platform. It also provides the environment for orchestrating the proofs.

The theorem prover includes an integration interface that provides the communication protocol for the verification tools. It consists of several parts: a datatype for all logical and control data transferred between tools, a datatype for the results of remote calls and support for installing and calling procedures, and a low-level communication manager.

The verification environment uses HOL98, a modern descendent of HOL, as the theorem prover component. HOL98 is a higher order logic theorem prover. Its logic is built on the predicate calculus of ML style typed system. Higher order logic is used as the glue logic to connect and integrate formal components. HOL's command language, ML, allows a developer to have a full programming language available in which to develop custom verification procedures. The tools integration interface library in HOL is provided by PROSPER.

3.2 Symbolic Simulator

The common design practice of validation by simulation has encouraged us to choose the ACL2 theorem prover as a component for the verification environment. ACL2 offers the capability of simulating test vectors and performing symbolic simulation. An interface (ACL2PII [25]) for PROSPER has been developed to allow results from ACL2 to be interpreted in HOL. ACL2PII is a dynamic link for translating theorems between two *live sessions* of HOL and ACL2, with communications going in both directions. The interface also allows a user to run ACL2 from within HOL.

The ACL2 and HOL theorem provers use different languages and different logics. ACL2 uses untyped s-expressions [27] to represent first order logics, whereas the HOL system uses typed terms for higher order logic. The interface implements a scheme for translating ACL2 s-expressions into HOL terms. A set of basic translation has been implemented so that the appropriate s-expressions can be automatically translated into booleans, natural numbers, integers, simple arithmetic expressions, characters, strings, lists and tuples. The interface also provides an environment to extend and add new translation clauses for new ACL2 theories.

Logically, ACL2 is being used as an *axiom–server* for facts about constants that are uninterpreted in HOL but have definitions in ACL2. The consistency of the axioms are assured by proofs being conducted in ACL2. This way of connecting ACL2 and HOL is pragmatic, but sound for the purposes of our application. The automatic transformation reduces the possibility of inconsistency when importing definitions and theorems from ACL2 into HOL.

3.3 Model Checker

The HOL98 distribution includes an early version of McMillan's SMV symbolic model checker as part of the temporal logic library. The model checker is embedded in HOL as one of the decision procedures for HOL's tactic language. Using this library, temporal properties specified in LTL notations can be validated in two ways, either by proving the properties using HOL tactics or by using the external model checker. When the model checker is used and the formula can be verified then the result from the model checker is represented as a HOL theorem using HOL's oracle mechanism. If the model checker reports an error, then a counterexample is provided. A detailed description of the embedding of LTL in HOL is presented in [22]

We replace the SMV model checker with the latest version from Cadence. This re–implementation of SMV uses LTL instead of CTL. Although for backward compatibility it supports CTL, the developers suggest to use LTL to achieve maximum performance. We extended the temporal library so that it is possible to use the Cadence SMV model checker with LTL notations. We embed a subset of SMVL in HOL using the deep embedding technique. In a deep embedding, the semantics of the language is constructed and an interpretation of the language is provided. This makes the system more modular. Previously, when we used model checker we had to specify formal models and properties which are to be verified in HOL. Now, we can define and verify the SMV model on its own and then automatically import the proved properties as HOL theorems.

4 Case Study: AMBA Bus Protocol and ARM7 Processor Based Verification Platform

We use RAPIER to describe our experience in the development of a reusable SoC verification platform. RAPIER is an integration platform architecture developed by the Institute for System Level Integration (ISLI) in Scotland [20]. It is based on the ARM *Advanced Microcontroller Bus Architecture* (AMBA) [3]. The platform contains an *Advanced High–Performance Bus* (AHB) and an *Advanced Peripheral Bus* (APB), an external memory controller, two timers, a UART, an Interrupt Controller, a System Controller, a system watchdog, a general purpose I/O block, five AHB masters, four AHB slaves, and four APB slaves. The AHB bus is the processor local bus (PLB) and the APB is the on–chip peripheral bus (APB). The architectural block diagram of the RAPIER platform is shown in Fig. 2.

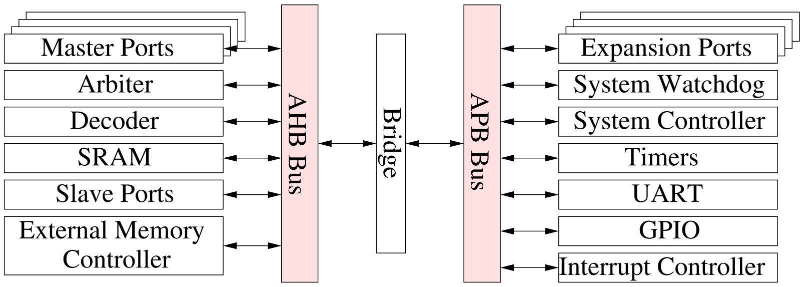

Fig. 2. Block diagram of RAPIER Platform.

4.1 AMBA Bus Protocol

AMBA is an on–chip bus specification that defines interconnection, communication, and management of functional blocks for SoC design. It is a technology independent specification. This ensures that the modules are reusable across diverse IC processes and technologies. It encourages standardise modular system design using a common bus protocol. This enhances the reuse design methodology for the modules.

Typically, AMBA based SoC design contains a high performance bus system such as the AHB. The bus is capable of handling high–bandwidth communication with the external memory interface, processor, on–chip memory, *Direct Memory Access* (DMA) module, and a bridge to the lower speed bus APB, where most peripherals in the system are located. An SoC system can have one or more masters. A typical system contains at least one processor. A DMA controller or a *Digital Signal Processor* (DSP) are also standard bus master devices. The external memory interfaces, on–chip memory, and APB bridge are typical AHB slaves. Most of the peripherals can be part of the system as AHB slaves, but more likely they are part of the AMBA APB.

4.2 The Protocol

In this case study, we partition the RAPIER platform and use only the AHB module. RAPIER's AHB contains an arbiter with five master ports and one slave port. We develop reusable properties for this module. The properties give the operational conditions for the system and the input/output behavioural relations.

The AMBA AHB process starts when a master asserts a bus request signal (req_m_i) to AHB arbiter. Then the AHB arbiter performs the arbitration process to determine which master is granted ($grant\_m_i$) the access to the bus. The granted master starts the data transfer process by sending control signals and an address. Control signals provide information about the transfer. One of these control signals is the type of transfer (HTRANS). There are four types of transfer: IDLE (mst_i_idle), BUSY (mst_i_busy), NONSEQ (mst_i_nonseq), and SEQ (mst_i_seq). In an IDLE transfer, the active master does not perform any data transfer. BUSY transfer is similar to IDLE, but it also indicates that the active master is inserting an IDLE cycle in the middle of a BURST operation. NONSEQ and SEQ signals indicate the control signal and address relation between current transfer and the previous one.

When the active master has started the transfer, the selected slave will respond with information on the transfer using HREADY and HRESP signals. Whenever slaves need to assert one or more wait states, the HREADY (*slv_ready*) signal is set to LOW. The HRESP signal is used to determine the status of transfer. There are four possible HRESP responses: OKAY (*slv_ok*), ERROR (*slv_error*), RETRY (*slv_retry*), and SPLIT (*slv_split*). The OKAY response indicates the slave's transfer is progressing without any problem. The ERROR response indicates that an error has occurred during the transfer. The RETRY response indicates that the transfer is not finished yet and the bus master has to retry the transfer until it is completed. The SPLIT response indicates that the transfer is not completed successfully; the bus master must retry the transfer when it is next granted access to the bus. In a SPLIT condition, the slave takes the responsibility to initiate a request to access the bus when the transfer can be completed.

The arbiter manages the arbitration processes. It monitors requests to access the bus from masters and to complete the split transfer from slaves. Then it decides which master has the highest priority to be granted the access. The arbiter is also responsible for ensuring that at any time there is only one master is granted access to the bus.

SMV is used to verify the implementation of AHB bus protocol. SMV uses the SMV Language (SMVL) and Verilog as its modelling languages. When the source is in Verilog, it needs to be translated into SMVL before it is model checked. The translation from Verilog to SMVL is done by using the translation tool *vl2smv*. The tool comes as part of the Cadence SMV distribution.

RAPIER AHB is implemented in Verilog. We needed to slightly modify the code to satisfy our system level verification methodology, which considers all components are black box components. The black box approach does not allow the use of any internal nodes or signals. One approach to overcoming this requirement is by bringing all internal nodes needed in the verification to the output interfaces. We also need to do abstractions to the data–bus and address–bus of the bus protocol to eliminate one source of explosion in BDD sizes during formal verification. The n-bit bus is replaced with a special scalar–set datatype called num [11].

4.3 ARM7 Processor

ARM7 is a 32–bit microprocessor from Advanced RISC Machines (ARM) [2]. It is based on the Reduced Instruction Set Computer (RISC) architecture. The processor features a three–stages pipeline architecture. Typically, in one cycle one instruction is being executed while the next one is being decoded, and the one after that is being fetched.

In our platform the formal model of the ARM7 processor is specified in LISP, the programming language of ACL2. Implementing the processor in LISP enables the model to be used in classical simulation test by executing the functional model with input test vectors. In the ACL2 environment, the LISP model is used as a formal model which enables user to perform symbolic simulation.

ARM7 is modelled as a finite state machine at the Micro–Architectural (MA) level. The model is a clock–cycle accurate model of the pipeline machine implementation. Every internal state transition corresponds to a hardware clock cycle. The MA is modelled using a state function, which is a mapping of *(f: inputs → state → state)* [18]. The *inputs* argument are the input interfaces of the processor. The *state* defines the internal state of the processor at a given time. It contains a list of all state–holding components of the processor, such as the registers, and flags. Our processor model does not feature Thumb and co–processor instructions.

The bus protocol and the processor are the core components of the verification platform. The platform is used in the development of an application, the Ethernet Switch. The system uses only two master modules.

5 Formal Properties

In this section, we explain the development of a verification platform for the system just described. The platform is based on specifications for the AMBA AHB bus protocol and the ARM7 processor. We then use these to develop platform specifications for two AHB masters. We require that the resulting system should have certain liveness properties.

The development stages are as follows: first, we establish generic properties of the platform which are based only on RAPIER's AMBA protocol properties. Second, we develop properties of the processor. Third, we integrate the bus arbiter and the processor by combining their properties. The result of this combination is used to define specifications for the remaining components. In effect, these specifications are the test–benches to define components' compatibilities with the system.

5.1 AMBA AHB Properties

Our verification platform is built around the AMBA bus protocol. In this first stage mentioned above, we define the environmental constraints for the protocol. These constraints provide operational conditions whereby the expected behaviours of the protocol are reached or proven correct. Conditions are proved using the SMV model checker; then they are imported into HOL98 as theorems (axioms).

All verification has been performed using a Linux machine with an Intel Xeon 2.4GHz processor with 3G RAM. The time used to import theorems from SMV into HOL is negligible compared to the time used to model check. Model checking of Theorems 1, 2, and 3 took approximately 25 seconds, 25 seconds, and 59 minutes of CPU time respectively.

From the documentation [13], we learn that the request from all masters connected to the AMBA AHB can be activated or de–activated by setting the clock blocking signals (*clocken_m*$_i$). When a master is de–activated, the clock signal is blocked for that master. Consequently, its requests to access the bus will be ignored and no grant signal can be assigned to it. To achieve maximum coverage of all masters' activities, all masters need to be activated. This is done by setting off the blocking control for the input clock of each module (*clocken_m*$_i$) with a HIGH signal.

One way to assure behavioural consistency of the system is by applying an initialisation sequence. This is achieved by triggering the reset signal. The environmental constraint express this by saying that the reset signal is active for at least one cycle and no reset is applied afterwards. We only analyse the behaviour of the model after the system is reset and all masters are active. This constraint is defined in *Assumption 1.*

Assumption 1.
$Reset \wedge XG\ \neg Reset \wedge G(\bigwedge_{1 \le i \le 5} clocken\_m_i)$

In SMV, *Assumption 1* is declared as a fairness condition for the system. The SMV code for this fairness condition is *SMV_Assumption1.* The fairness properties are enforced by assuming them to be true, using the SMV *assume* construct.

```
SMV_Assumption1:
    assert (Reset & XG ~Reset & G(clocken_m1 &
            clocken_m2 & clocken_m3 & clocken_m4));
assume SMV_Assumption1;
```

The AHB arbiter receives requests from up to five AHB masters. It then uses a fixed priority rule to determine which master should be granted bus ownership. Master m_5 is assigned the highest priority and master m_0 the lowest priority. If no master is requesting the bus, then unless m_1 is in the split mode [3], bus ownership is granted to m_1. If default master (m_1) is in split mode, the bus is granted to dummy master (m_0). The AHB arbiter has the responsibility to ensure that at any time only one master is being granted bus ownership. The arbiter also guarantees that at any time there is one master which is granted the bus. This is shown in *SMV_Theorem1*. The mutual exclusion properties are described in *SMV_Theorem2*.

```
SMV_Theorem1:
    assert G(grant_m0 | grant_m1 | grant_m2 |
             grant_m3 | grant_m4 | grant_m5);
using SMV_Assumption1 prove SMV_Theorem1;

SMV_Theorem2:
    assert G(~(grant_m0 & grant_m1) &
             ~(grant_m0 & grant_m2) &
             ...
             ~(grant_m4 & grant_m5));
using SMV_Assumption1 prove SMV_Theorem2;
```

SMV_Theorem1 and *SMV_Theorem2* are proved using the fairness condition *SMV_Assumption1*. We instruct SMV to use the constraints by a **using** *assumptions* **prove** *theorems* statement.

The interface between SMV and HOL enables users to automatically import properties proved in SMV into HOL. The interface analyses the SMV code to find relevant information about the properties being verified. It also gathers which components or modules and assumptions are used in the verification. The modules and assumptions become the antecedents and the properties being proved as the conclusions of implications in HOL. For example, SMV_Theorem1 is imported into the HOL environment by using the command *get_smv_theorem*. The HOL theorem is:

HOL_Theorem1:
$(\text{AHB} \wedge \text{Reset} \wedge \text{XG} \neg\text{Reset} \wedge$
$\quad \text{G}(\text{clocken\_m}_1 \wedge \text{clocken\_m}_2 \wedge \text{clocken\_m}_3 \wedge \text{clocken\_m}_4))$
$\rightarrow$
$(\text{G}(\text{grant\_m}_0 \vee \text{grant\_m}_1 \vee \text{grant\_m}_2 \vee$
$\quad \text{grant\_m}_3 \vee \text{grant\_m}_4 \vee \text{grant\_m}_5) \wedge$
$\ \text{G}(\neg(\text{grant\_m}_0 \wedge \text{grant\_m}_1)\wedge$
$\quad \neg(\text{grant\_m}_0 \wedge \text{grant\_m}_2)\wedge$
$\quad \ldots$
$\quad \neg(\text{grant\_m}_4 \wedge \text{grant\_m}_5))$

Another style of *HOL_Theorem1* is presented in *Theorem 1*. The theorem says that when AHB is initialised with conditions described in *Assumption 1*, there will be exactly one master being granted bus ownership.

Theorem 1.

$$(\mathbf{AHB} \wedge \mathit{Assumption(1)}) \rightarrow (G(\bigvee_{0\leq i,j\leq 5} \mathit{grant\_m}_i) \wedge G(\bigwedge_{0\leq i,j\leq 5, i\neq j} \neg(\mathit{grant\_m}_i \wedge \mathit{grant\_m}_j)))$$

For the reminder of Section 5.1, we use the notations employed in *Assumption 1* to describe the SMV fairness constraints and *Theorem 1* to describe SMV theorems when they are imported into HOL.

After defining the initialisation process, we need to learn about the specific behaviour of the system. The resources we have for this are the documentation and the circuit itself. In most cases, however, the existing documentation is not detailed enough to provide the specific information needed. Furthermore, the system may come as a black box system where minimum information of the circuitry are available. One approach that can be taken is by performing experimental verifications using the documented specifications as the guidelines. In our case, we use SMV to learn about our AHB system. When incorrect constraints are used in the verification, SMV generates a counter example. We use the documentation and feedback from SMV to determine the operational conditions of the system that can lead to the expected behaviours.

One of master's behaviours is that it can request the arbiter to perform a burst process or a lock process. When the arbiter allows the master to perform those processes, the arbiter state machine goes into either burst mode or lock mode state. In these states, the system goes into an internal loop and continues to grant the bus to the active master until the process is finished. The only exception is when the arbiter goes into a lock–split state, which forces the arbitration to grant the bus to the dummy master until the split process is completed. Lock–split state is a condition when the arbiter is serving active master lock request, slave responds with a split signal. The arbiter starts a new arbitration when the process in burst mode or lock mode is completed.

At this stage, we need to find the general conditions that ensure that all process modes can be completed. When a master is granted the bus, the completion of the process depends on the response from the slave. The slave informs the arbiter and the master that the data is ready by emitting a *slv_ready* signal. We assume slaves have the fairness property of eventually responding to any request. At the same time, the master must be able to acknowledge the slave response. This requires a condition where if master m_i is granted the bus then eventually the active master is not in a busy mode and slave issues a ready signal. This fairness constraint is described in *Assumption 2.*

Assumption 2.
$GF\ slv\_ready \wedge G(\bigwedge_{1\le i,j\le 5} grant\_m_i \rightarrow XF(\neg mst_i\_busy \wedge slv\_ready))$

The transition of the arbiter's state machine into lock mode can be observed from the response on grant and lock signals of each masters. When an active master is sending a lock signal, then the arbiter will go to lock mode. We define this condition as $(\bigvee_{1\le i\le 5} (grant\_m_i \wedge lock\_m_i))$ and abbreviate it as the *lock_req* signal. When *lock_req* goes HIGH then the arbiter will be in the lock mode. Whenever the system enters a lock mode, there is a possibility that the system is trapped and has reached a deadlock condition. We prevent this condition by stating that every master which asserts a lock signal will eventually de–assert it.

There is also a possibility that the lock mode operation goes into an alternating sequence in which the master sends the lock/unlock signal and the slave sends the split/retry-ok/error signal. For examples, everytime active master de–

assert lock signal, slave responds with split/retry signal. This condition forces arbiter to go back into lock or lock–split mode state. If this condition always occurs, the system will be trapped in the lock mode. We choose to allow this condition to happen and examine the possible sequences needed to break this loop–trap. The requirement to exit from the loop–trap is described in *Assumption 3*.

Assumption 3.

$$\begin{aligned}&G\ (lock\_req \rightarrow F\ \neg lock\_req) \wedge \\ &GF\ (lock\_mode \rightarrow (\neg grant\_m_0 \wedge slv\_ok/slv\_error \wedge \neg lock\_req \wedge \\ &\qquad X(slv\_ready \wedge slv\_ok/slv\_error \wedge \neg lock\_req)))\end{aligned}$$

The description of the above assumption is as follows: first, when the active master asserts a lock signal, it will eventually de–assert it. Second, lock mode will always be terminated after two cycles. In the first of these cycles, it is required that the bus is not granted to m_0. In the second cycle, the slave module has to acknowledge that it is ready to complete the transfer. In both cycles, the master has to be able to retract the lock signal (unlock), and the slave must not issue a split or retry response.

A new arbitration is achieved when the system is in burst mode or able to exit from the lock–trap while in lock mode. This condition is indicated by *new_cycle* signal. When HIGH output on this signal indicates that the arbiter is performing a new arbiration process. The exit requirements are defined in *Assumption 1,2*, and *3*. Assuming the exit requirements are fair, we prove that the arbiter will always eventually perform a new arbitration. The theorem is described below:

Theorem 2.

$$(\mathbf{AHB} \wedge Assumption(1,2,3)) \rightarrow GF\ new\_cycle$$

There is a possibility that a granted master is forced by a slave into split mode. When this condition occurs, the arbiter memorises which master has been split using the *split_m*$_i$ signal. In this case, arbiter will ignore all incoming requests from master m_i until it receives *un–split* signal from the slave. The *un–split* signal indicates that the data for the master is ready for transfer. To avoid the scenario that a master remains in split mode indefinitely, we define a new fairness condition described in *Assumption 4*.

Assumption 4.

$$G \bigwedge_{1\leq i\leq 5} (split\_m_i \rightarrow F\ un\text{–}split\_m_i)$$

Every clock cycle, the arbiter evaluates the latest input signals and decides what action it will take. When a request send by a master module is not granted, the module needs to keep requesting. This is because the arbiter does not memorise any incoming signals. If the master retracts its request signal, the arbiter will assume the corresponding master has cancelled its request.

The arbiter uses a fixed priority scheme to decide which master is granted access to the bus. The fixed priority scheme will always prevent any lower priority master being granted bus ownership. We need to create a situation where the possibility of granting control to this master exists. A request from m_i can only be granted whenever no higher priority master is sending a request signal or when the higher priority master is in split mode. The request constraints are described in *Assumption 5.*

Assumption 5.

$$G \bigwedge_{1\leq i\leq 5} ((req\_m_i \wedge X\, \neg grant\_m_i) \rightarrow X\, req\_m_i) \wedge$$
$$G \bigwedge_{1\leq i\leq 4} (req\_m_i \rightarrow F\, (\bigwedge_{i<j\leq 5} (\neg req\_m_j \vee split\_m_j) \wedge X\, new\_cycle))$$

After we successfully create the general scenario for a new arbitration, we can use it to obtain the requirements for the arbiter to grant every incoming master request. The additional rules are described in Assumption 4 and 5. The general request-grant theorem is described in *Theorem 3.* The theorem says that every master request will eventually be granted, provided all requirements defined in *Assumption 1* through *Assumption 5* are satisfied:

Theorem 3.

$$(\mathbf{AHB} \wedge Assumption(1,2,3,4,5)) \rightarrow G\, (\bigwedge_{1\leq i,j\leq 5} req\_m_i \rightarrow F\, grant\_m_i)$$

Theorem 3 defines only the liveness condition of every master's request. In order to guarantee liveness of the system, all constraints must be satisfied. This means that all masters have to operate fairly so that every master has the chance to access the bus. In Section 5.3, we describe how we refine *Theorem 3* to construct an application specific verification platform.

5.2 ARM7 Properties

ARM7 processor is the second core component of the verification platform. In AMBA AHB, processor is defined as the default master and connected to the ports of m_1. The processor is modelled in ACL2 using functional modelling style. In this style, the output signals of a component are given as a function of the input signals.

ARM7execute is a single–step execution function for the ARM7 processor. The function takes five arguments. The first is the input signal for reset. The second is the signal from the arbiter to grant the access to the bus. The third argument is the interrupt input signal. The fourth is the data–in from the AHB bus. The last argument is the internal state function of the processor. Evaluating *ARM7execute* will compute the updated initial internal state and return this updated state.

Similar to the bus protocol, we also need to obtain properties of the processor. They are obtained by proving facts using the ACL2 theorem prover. In this paper, we describe three features of the processor that have been verified.

All analysis was performed under the condition that no reset is applied to the processor. The processor's properties are as follows:

- A *busy* signal is emitted only when the processor is executing co–processor instructions. Since our processor model does not implement co–processor instructions, it will never send a *busy* signal.
  $\neg reset \rightarrow (\neg\ Pbusy\ (ARM7execute\ 0\ grant\ interrupts\ data\ Pstate))$

- The processor will continue its evaluation only when it recieves a grant signal. If it does not, then it goes to an idle state and maintains its internal state. This means that the processor holds its request signal whenever it is not granted.
  $(\neg reset \wedge \neg grant) \rightarrow (ARM7execute\ 0\ 0\ interrupts\ data\ Pstate) = Pstate.$

- The ARM7 processor is capable of performing a lock sequence. We prove that after at most three execution cycles, the processor will release the bus.

$$\begin{aligned}&(P1 = (ARM7execute\ reset0\ grant0\ interrupts0\ data0\ P0) \wedge\\ &\ P2 = (ARM7execute\ reset1\ grant1\ interrupts1\ data1\ P1) \wedge\\ &\ P3 = (ARM7execute\ reset2\ grant2\ interrupts2\ data2\ P2)) \rightarrow\\ &((\neg reset0 \wedge grant0) \rightarrow\\ &\ (\neg Plock(P1) \vee ((\neg reset1 \wedge grant1) \rightarrow\\ &\qquad\qquad (\neg Plock(P2) \vee ((\neg reset2 \wedge grant2) \rightarrow \neg Plock(P3))))))\end{aligned}$$

In our methodology, all components are combined and integrated in HOL. They are specified as relational models in higher order logic by defining predicates that state which combinations of values can appear on their external ports. When a component is defined as a functional model, as is the case with our ACL2 model of ARM7, it needs to be transformed into a relational one. A wrapper is created to bridge the functional model and the relational model.

The ACL2 processor function *ARM7execute* is transformed in HOL into the relational model called *ARM7*. The relational model of the processor is defined as follows:

$$\begin{aligned}\mathrm{ARM7} \overset{def}{=}\ &(\mathrm{Pst}_0 = \mathrm{P}_0) \wedge\\ &(\mathrm{Pst}_{(t+1)} = \mathrm{ARM7execute\ reset\ grant\ interrupts\ data\ Pst}_t)\end{aligned}$$

Pst is a function from time to the processor's state. The index subscript to Pst indicates the relative time at which the state occurs. Pst_0 is the state of the processor at time 0 and P_0 is the initial state of the processor. As discussed above, ACL2 theorems for the processor are automatically imported into HOL as trusted axioms. A small amount of very simple theorem proving is needed to simplify the HOL properties obtained from ACL2 theorems. The final HOL theorem is as follows:

Theorem 4.

$$(\mathbf{ARM7} \wedge (\mathrm{G}\ \neg\mathrm{reset})) \rightarrow$$
$$((G\ (\neg mst_1\_busy) \wedge$$
$$G\ (\neg grant\_m_1 \rightarrow (Pst_{(t+1)} = Pst_t)) \wedge$$
$$G\ (grant\_m_{1(t)} \rightarrow (\neg Plock(Pst_{(t+1)}) \vee (grant\_m_{1(t+1)} \rightarrow$$
$$(\neg Plock(Pst_{(t+2)}) \vee (grant\_m_{1(t+2)} \rightarrow \neg Plock(Pst_{(t+3)}))))))$$

5.3 Application Specific Platform

RAPIER is an environment used in teaching at the ISLI. One application case study was to build an Ethernet Switch using the platform. The Ethernet Switch system uses two AHB masters: the ARM7 processor and a memory controller. In this platform, all slaves are required to give an immediate response for any master's request. The slaves are not allowed to respond with a split or retry signal. Our goal is to find the specifications or requirements for the memory controller and the slaves so that all desired properties are satisfied.

Interconnection of the components in the verification platform is a straightforward step. The formal models are connected and integrated with logical conjunction in higher order logic. The integration of the AHB bus protocol and the ARM7 processor are just defined as (**AHB** $\wedge$ **ARM7**).

We set our goal to have a system which has liveness properties. In this condition, all requests are always granted. *Theorem 3* shows the general rules or constraints for granting master's requests. We use these constraints to define the specifications of a system which has the desired liveness properties.

The Ethernet Switch system uses only two masters. The other masters are left inactive. This fact is the new constraint for the AHB bus. We use this constraint to refine existing AHB properties. The refinement is performed either using the model checker (SMV) or the theorem prover (HOL). In either case, we use existing properties and simplify the constraints of the AHB bus protocol. We do not need to re–model check the bus protocol from scratch for the system with two masters. We choose to import all proofs about AHB into HOL where we perform system level integration and verification.

The non–existence of m_3 to m_5 means that there is no request from any of these modules. One of the implications of this is that no grant signals are ever sent for these masters. The slave requirement of not allowing *split* or *retry* means a slave can only respond with *ok* or *error*. Because a slave is never emitting a *split* signal, no split condition will ever occur. In SMV we prove the system has these properties. The properties are used as the refinement constraints to simplify the generic properties of AMBA AHB. These constraints are defined as follows:

$$\bigwedge_{3 \le i \le 5} (G\neg req\_m_i \rightarrow G\neg grant\_m_i) \wedge$$
$$(G\ slv\_split/slv\_retry) \rightarrow G(\bigwedge_{1 \le i \le 5} \neg split\_m_i \wedge \neg grant\_m_0 \wedge slv\_ok/slv\_error)$$

The constraints eliminate the need for *Assumption 4*. They also simplify *Assumption 1,3,5* with *Assumption 6,7,8* respectively. The new assumptions eliminate all properties related to m_0, m_3, m_4, m_5, and slave *split/retry* response.

The clock enable signals in Assumption 1 are only needed when they are used. If there is no master connected to the corresponding port, the condition of these signals can be ignored or turned off. The new assumption is shown below:

Assumption 6.
$Reset \wedge XG\ \neg Reset \wedge G(\bigwedge_{1 \le i \le 2} clocken\_m_i)$

The restriction on slave modules not allowing them to send split or retry signals reduces *Assumption 3* dramatically. It eliminates the need to include a dummy master module. Furthermore, the exit constraints when the arbiter is in lock mode depend only on the slave's ready signal and master's lock request signal. The reduced fairness constraints are as follows:

Assumption 7.
$G\ (lock\_req \rightarrow F\ \neg lock\_req) \wedge$
$GF\ (lock\_mode \rightarrow (\neg lock\_req \wedge X(slv\_ready \wedge \neg lock\_req)))$

The properties of *Assumption 5* are reduced to m_1 and m_2. In the specialized platform, the system's liveness constraints only depend on the fairness condition of m_2 not infinitely requesting the bus. Because m_1 is the default master, when m_2 does not request the bus, arbiter will always grant the bus to the default master. The simplified assumption is described in *Assumption 8.*

Assumption 8.
$G \bigwedge_{1 \le i \le 2} ((req\_m_i \wedge X\ \neg grant\_m_i) \rightarrow X\ req\_m_i) \wedge$
$G\ (req\_m_1 \rightarrow F(\neg req\_m_2 \wedge X\ new\_cycle))$

The behaviour of the ARM7 processor is given by *Theorem 4*. One of the properties is that the processor never sends a *busy* signal. This fact eliminates the dependency of *Assumption 2* on the processor's behaviour. The new constraints are given in *Assumption 9.*

Assumption 9.
$GF\ slv\_ready \wedge G(grant\_m_2 \rightarrow XF(\neg mst_2\_busy \wedge slv\_ready))$

When the processor is in a wait state, it maintains all of its properties. This means when the processor sends a request signal and the arbiter tells the processor to wait, the processor will keep sending the request signal. This property refines *Assumption 8* into *Assumption 10.*

Assumption 10.
$G\ ((req\_m_2 \wedge X\ \neg grant\_m_2) \rightarrow X\ req\_m_2) \wedge$
$G\ (req\_m_1 \rightarrow F(\neg req\_m_2 \wedge X\ new\_cycle))$

Theorem 4 also shows that when the processor is locking the bus, it will eventually unlock it in at most three execution cycles. The arbiter also guarantees that in lock mode the active master always keeps the bus. These conditions refine *Assumption 7* to *Assumption 11.*

Assumption 11.
$G((grant\_m_2 \wedge lock\_m_2) \rightarrow F(grant\_m_2 \wedge \neg lock\_m_2)) \wedge$
$GF\ (lock\_mode \rightarrow (\neg lock\_req \wedge X(slv\_ready \wedge \neg lock\_req)))$

Finally, the Ethernet Switch platform is defined in *Theorem 5.* It says the platform has two masters. It is constructed from the AHB bus protocol and the ARM7 processor. When the system is initialised with the sequence described in *Assumption 6* and the constraints described in *Assumption 9,10,11* are satisfied, the system will always provide fair services for its two masters. Light weight theorem proving is needed to prove *Theorem 5.*

Theorem 5.
$(\mathbf{AHB} \wedge \mathbf{ARM7} \wedge Assumption(6,9,10,11)) \rightarrow$
$G\ (req\_m_1 \rightarrow F\ grant\_m_1 \wedge req\_m_2 \rightarrow F\ grant\_m_2)$

Based on *Theorem 5*, we can analyse the requirements and define the specifications for each module. The second master (memory controller) has to satisfy the following specifications:

– The module has to be capable in maintaining its request signal until it is granted.
– The module has to be able to accept a response from a slave by not always engaging in a busy mode.
– If the module is capable asserting a lock signal, it has to be able to de-assert it until a new arbitration cycle is reached.
– In order to let a lower priority master access the bus, the module should not infinitely request the bus. One way to achieve this is by introducing one additional rule: every completed request sequence must be followed by a sequence of idle states. In this way, the system can guarantee that all requests can be served.

The slaves in this platform have to satisfy specifications as follows:

– By definition, all slaves are not allowed to send a retry or split signal.
– They have to be able to respond to all requests.
– To prevent any erratic behaviours of the slave, we define one additional rule which controls the behaviour of the slave: when all input are stable, the output of the slaves will eventually become stable. This means that when a slave is ready to respond to a master's request, afterwards the slave's output remains stable as long as the input does not change.

In this methodology, we obtain specialised specifications for both master and slave modules. This specifications feature tighter requirements in comparison to the standard ones. The specifications are geared to satisfy the application specific requirements. Designing the modules under these specifications guarantee the system to fulfil the application's specific requirements.

6 Conclusions and Future Work

We have presented a tool architecture and methodology to perform formal verification for system on chip designs. The verification environment combines various formal tools which enable verification engineers to perform symbolic simulation, model checking, and theorem proving. The mechanism for sharing information reduces the possibility of errors being made during the translation of theorems from one formal tool to the other. It also allows each component to be modelled in the most suitable formalism.

The methodology is based on the development of a generic formal verification platform in which applications can be developed. The generic platform behaviours are described as a set of formal properties. The generality of the properties make them reusable in the development of platform specific applications. The properties can be used to develop the specifications of the components of the platform. They can also be used to analyse the behaviour of the platform with a set of components.

We have developed a standard integration platform containing the AMBA–AHB bus protocol and a ARM7 processor. We described the development of reusable formal properties for this platform. The properties define the generic behaviour of the system. We used this platform to build an application. By evaluating the platform's properties with the application requirements, we obtain the specification for the remaining components.

Our future research will build a more comprehensive verification platform on top of our proof environment. The platform will be based on the full specification of the AMBA bus protocol and the ARM7 processor. We are aiming for a 'plug and play' verification environment, involving a collection of reusable proofs. The verification platform will enable the possibility to be used as a workbench to develop detailed specifications.

Acknowledgements

The authors thank the Institute for System Level Integration for providing the RAPIER platform for this work, the Veriscope research group for the use of their machine, Michael Dales for helpful comments on a draft of this paper, and the reviewers for their feedback.

This work has been supported in part by SHEFC RDG grant 85, *Design Cluster for System Level Integration*, and the ESPRIT PROSPER project LTR 26241.

References

1. Mark D. Aagaard, Robert B. Jones, and Carl-John H. Seger. Combining Theorem Proving and Trajectory Evaluation in an Industrial Environment, in *the 35th Design Automation Conference*, San Francisco, California, June 1998, pp. 538–541.
2. ARM, ARM-7 Datasheet, DDI 0020C, December 1994.

3. ARM, AMBA specification ver 2.0, IHI-0011A, May 1999.
4. Mark Birnbaum and Howard Sachs, How VSIA Answers the SOC Dilemma, *IEEE Computer* magazine, June 1999, pp. 42–50.
5. A.J. Camilleri, A Hybrid Approach to Verifying Liveness in a Symmetric Multi-Processor, Eds Elsa L. Gunter and Amy Felty in *Theorem Proving in Higher Order Logic*, Murray Hill, New Jersey, August 1997, Springer-Verlag LNCS 1275, pp. 49–67.
6. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. *Surviving the SOC Revolution. A Guide to Platform-Based Design*, Kluwer, 1999.
7. Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit, Eds S. Graf and M. Schwartzbach in *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000*, Berlin, March/April 2000, Springer-Verlag LNCS 1785, pp. 78–92.
8. M.J.C. Gordon and T.F.Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.
9. *The HOL System Description, HOL98 Taupo–6*, University of Cambridge, February 2000.
10. W.A. Hunt. *FM8501: A Verified Microprocessor*, 1994, Springer-Verlag LNCS 795.
11. C.N.Ip and D.L.Dill. Better Verification Through Symmetry, Eds D. Agnew, L. Claesen, and R. Compasano, *Computer Hardware Description Languages and their Applications*, Elsevier Science Publishers B.V., Amsterdam, Netherland, pp. 87–100.
12. R.B. Jones, J.W. O'Leary, C.-J.H. Seger, M.D. Aagaard, and T.F. Melham. Practical Formal Verification in Microprocessor Design, *IEEE Design & Test of Computers* magazine, July/August 2001, pp. 16–25.
13. Mark Litterick, ARM Integration Platform Power Management, The Institute of System Level Integration, Scotland, November 2001.
14. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning, An Approach*, Kluwer, 2000.
15. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning, ACL2 Case Studies*, Kluwer, 2000.
16. Michael Keating and Pierre Bricaud, *Reuse Methodology Manual For System–On–a–Chip Designs*, Kluwer Academic Publisher, Norwell Massachussetts, 1999.
17. Kenneth L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publisher, Norwell Massachussetts, 1993.
18. J Strother Moore, Symbolic Simulation: An ACL2 Approach. Eds Ganesh Gopalakrishnan and Phillip Windley in *Formal Methods in Computer-Aided Design*, PaloAlto, California, November 1998, Springer–Verlag LNCS 1522, pp.334–350.
19. Carl Pixley, Formal Verification of Commercial Integrated Circuits, *IEEE Design & Test of Computers* magazine, July/August 2001, pp.4–5.
20. RAPIER, The Institute of System Level Integration, Scotland, 2001.
21. Ann Marie Rincon, Cory Cherichetti, James A. Monzel, David R. Stauffer, and Michael T. Trick. Core Design and System-on-a-Chip Integration, *IEEE Design & Test of Computers* magazine, October–December 1997, pp. 26–35.
22. K. Schneider. Yet another look at LTL model checking. Eds Laurence Pierre, and Thomas Kropf, in IFIP WG10.5 Advanced Research Working Conference on *Correct Hardware Design and verification Methods*, Bad Herrenalb, Germany, September 1999, Springer–Verlag LNCS 1703, pp.321–325.

23. Kanna Shimizu, David L. Dill, and Ching-Tsun Chou. A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol. Eds Tiziana Margaria, and Tom Melham, In *Correct Hardware Design and verification Methods*, September 1999, Springer–Verlag LNCS 2144, pp.340–354.
24. Madayam Srivas, Harald Rueβ, and David Cyrluk. Hardware Verification using PVS in *Formal Hardware Verification Methods and Systems in Comparison*, edited by Thomas Kropf , July 1997, Springer Verlag LNCS 1287, pp: 156–205.
25. Mark Staples, *Linking ACL2 and Hol*, Computer Laboratory, University of Cambridge, Technical Report No. 476, November 1999.
26. Kong Woei Susanto, An integrated Formal Approach for System on Chip, In *IP based Design*, Grenoble, France, October 2002, pp: 119–123.
27. Patrick Henry Winston and Berthold Klaus Paul Horn, *LISP*, Addison–Wesley Pub.Co., 1989.