A Verification Platform for System on Chip

Kong Woei Susanto

A Dissertation submitted to the University of Glasgow in partial fulfillment of the regulations for the degree of Doctor of Philosophy

October 2003

Department of Computing Science
University of Glasgow
Glasgow, UK

Abstract

System on Chip technology will reshape common design practice. The pressure to create a working System on Chip design as early as possible leads designers to consider using a platform based design method, called a system integration platform. In this design methodology, a system is built from intellectual property blocks in a *plug and play* environment. By using this approach, designing an application is a matter of selecting from a set of standard components with compatible specifications. Subsequently, a similar platform can be constructed for formal verification. Every component in the integration platform has a corresponding formal model in the formal platform.

This dissertation proposes a methodology to develop formal models of System on Chip design and to verify the design from a system level point of view. The methodology is based on two components: a formal verification environment and a formal verification platform. A formal verification environment is an environment where a selection of formal tools are integrated to offer a complete set of formal verification techniques. It combines the capabilities of the HOL98 theorem prover, the ACL2 theorem prover, and the SMV model checker. A formal verification platform is a standardised platform where formal models can be integrated and system level validation of the design can be performed. In the verification platform, all formal models are formalised in the most suitable formalism. They are connected using higher order logic as the glue logic. The formal verification platform provides an environment to analyse the combined properties of the design.

Two formal verification platforms were formalised to demonstrate the application of a formal verification environment in system level verification. In the first platform, a complete system was formalised. The verification efforts were targeted to verify whether properties of the system as a whole satisfied the specifications. The verification showed that the system had liveness properties and that all master requests will eventually be granted by the arbiter. The software was also verified to ensure that it was correctly executed. In the second platform, a partial system is defined. Applications were developed by integrating additional components onto the platform. The formal verification platform was used to obtain specific properties of the system. These properties can be used as the guidelines for tighter specifications in the selection of components.

Declaration

The studies outlined in this dissertation were undertaken in the Department of Computing Science, University of Glasgow, under the supervision of Prof. Tom Melham, Dr. Simon Gay, and Dr. John O'Donnell. This dissertation has not been submitted at any other university. All of the work was performed by the author, except otherwise indicated.

The case studies presented in this dissertation have been published. The verification of *Simple Integration Platform* has been published in [77]. The development of *ARM verification platform* have been published in [78].

Kong Woei Susanto
Glasgow, October 2003.

In memory of my grandparents.

Acknowledgements

I would like to extend my gratitude and thanks to my supervisor Tom Melham for providing guidance and support throughout my PhD research. I am grateful for his patience in guiding me to understand the world of theorem proving. He has been a constant source for valuable technical feedback even after his departure from the department. I am overwhelmed with his kindness to my family during our time in Glasgow. Without his encouragement and support, this dissertation would not exist.

Special thanks must be given to Simon Gay and John O'Donnell, who kindly become my supervisors when Tom left the department. I would like to thank the formal analysis, theory, and algorithms groups for their constant support. Special thanks to Alice, Lyndell, Stephan, Peter, Claire, Michael, and Ashish for their valuable thoughts. I also like to thank J Moore, Jun Sawada, Mark Staples, Michael Norrish, Konrad Slind, and Richard Boulton for their help with the system.

Special thanks also to Prof. Steve Beaumont and the Institute for System Level Integration for providing the RAPIER platform for this work and the Veriscope research group for the use of their machine.

I would like to thank Intel Strategic CAD Labs for providing me an internship opportunity. Special thanks to Brian Moore and John O'Leary for guiding me during my internship in working on an industrial scale verification.

I am thankful for the financial support provided by the University of Glasgow studentship, the Overseas Research Studentship scheme, SHEFC RDG grant 85, *Design Cluster for System Level Integration*, and the ESPRIT PROSPER project LTR 26241.

Finally, I wish to thank my parents and brothers for their support. To my wife Kristin, for her love, patience, and support throughout my academic endeavour. To my daughter Stephanie, who fills my days with joy and happiness.

Contents

List of Figures

List of Tables

List of Definitions

List of Lemmas

List of Theorems

Glossary

ACL2	A Computational Logic for Applicative Common Lisp
ACL2PII	ACL2 PROSPER Integration Interface
AHB	Advanced High Performance Bus
AMBA	Advanced Micro-controller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced RISC Machines
ASB	Advanced System Bus
ASIC	Application Specific Integrated Circuit
ADD	Area Driven Design
BBD	Block Based Design
BDD	Binary Decision Diagram
BIST	Build In Self Test
CMOS	Complementary Metal Oxide Semiconductor
CPE	Core Proof Engine
CPSR	Current Program Status Register
CTL	Computational Tree Logic
DMA	Direct Memory Access
DSP	Digital Signal Processing
EDA	Electronics Design Automation
FP	Floor Planning
FSM	Finite State Machine
FVE	Formal Verification Environment
FVP	Formal Verification Platform
GDSII	Graphical Design System II
GPIO	General Purpose Input/Output
GPR	General Purpose Register
GSI	Giant Scale Integration
HCI	Host Controller Interface
HDL	Hardware description Language
IC	Integrated Circuit
I/O	Input/Output
IP	Intelectual Property
ISLI	Institute for System Level Integration

LSI	Large Scale Integration
LTL	Linear Time Logic
MOS	Metal Oxide Semiconductor
MSI	Medium Scale Integration
OCP	Open Core Protocol
OPB	On-chip Peripheral Bus
PBD	Platform Based Design
PC	Program Counter
PCB	Printed Circuit Board
PII	PROSPER Integration Interface
PLB	Processor Local Bus
RAM	Random Access Memory
ROM	Read Only Memory
RISC	Reduced Instruction Set Comnputer
RTL	Register Transfer Level
RTOS	Real Time Operating System
SBD	Schematic Based Design
SFR	Special Function Register
SIMBC	System Integration Module and Bus Control
SIP	Simple Integration Platform
SLI	System Level Integration
SoC	System on Chip
SPSR	Saved Program Status Register
SSI	Small Scale Integration
STA	Static Timing Analysis
TDD	Timing Driven Design
TTL	Transistor-Transistor Logic
UART	Universal Asynchronous Receiver/Transmitter
ULSI	Ultra Large Scale Integration
VC	Virtual Component
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC HDL
VLSI	Very Large Scale Integration
VSIA	Virtual Socket Interface Alliance

Chapter 1

Introduction

1.1 Background

Since the early 1950s, the complexity of transistor based electronics has exponentially grown at an immense rate. This growth has been achieved as a result of miniaturisation of transistor sizes. In addition to the growth in system complexity, the requirements on time to market for electronic products has become shorter. Time to market is the key prerequisite in the electronics industry.

In the near future, the microelectronics industry will face the reality of a billion transistors on a single chip. In this technology, a designer is able to implement a complete and complex system on a single chip. The technology is commonly known as *System on Chip* (SoC).

The SoC design concept follows traditional design flows. Since the early days, silicon and design automation technology has facilitated the integration of more functionality onto a single piece of silicon. However, with each advancement in silicon technology and the availability of transistors on a single die, new design challenges emerge. In the early days, designers were challenged to place as much functionality as possible onto a single die. Nowadays the number of available transistors exceeds the requirement to complete a full system in a single chip. Traditionally, designers focused on creating original design content and verifying it. The new approach shifts the content based approach to the compositional approach [38]. In the new approach, designers deal with evaluating, integrating and verifying the components.

The SoC revolution will bring this technology to a broader spectrum of users [36]. Furthermore, recent industry developments are setting the stage for more widespread use. The emergence of multi-core *System-Level Integration* (SLI) platform chips and the increasing standardisation of peripheral interconnect enables critical design efficiencies and standardised architectures. These efficiencies are critical elements both in the proliferation of electronic systems and in the use of SoC design methodology to create them.

1.2 Problem Definitions

Historically, *Hardware Description Language* (HDL) and *Schematic-Based Design* (SBD) flows were used in creating new logic circuits which implemented the necessary functionality [9]. Increasingly along with the advancement of the design and manufacturing technology, the *integrated circuits* (ICs) are comprised from a collection of *intellectual property* (IP) blocks or *virtual components* (VCs). These IP blocks are predefined, large grained blocks, such as filters, peripherals, memories, and processors, whose function has been precisely specified. They can be developed in-house or originated from external vendors. When IPs become widely available the design focus will shift to reuse rather than design. The SoC design is becoming much more a matter of design composition rather than of design creation.

The SoC design shifts the designer's design focus towards devising the following problems:

- How the functionality will be divided between hardware and software.
- How and what IP blocks will be utilised.
- How the system will be interconnected and validated.

This compositional or reuse based design methodology will be the problem addressed by SoC designers. A standardised platform and application-specific architectural context will play a major role in achieving the plug-and-play environment using reusable components. The verification of SoC design is arguably the biggest challenge for designers. On the other hand, designers are still constrained in using the traditional time consuming logic simulation methods for a full-chip validation. A new design methodology for SoC design is needed to address these problems. The new methodology has to be able to reduce the amount of analysis, debugging, and optimisation that takes place in the early development stages. A common co-validation environment for both hardware and software engineers will facilitate more productive interaction as well as a more optimised implementation. In addition, rapid verification using an abstract representation of IP models also helps to speed up the simulation. Finally, the use of mathematical techniques in formal verification will remove uncertainty, increasing the design confidence, and reducing verification time.

The formal verification methodology has recently been used on a regular basis to verify hardware specifications and models within the industrial community [25]. Based on the technology, formal hardware verification can be categorised in three groups: equivalence checker, model checker, and theorem prover [25, 71].

An equivalence checker uses a mathematical approach to verify the equivalence of a reference or golden model to the implementation of the model. A model checker compares a design to a set of logical properties of a design's behaviour, where the properties are a direct representation of the specification of the design. Finally, the theorem proving method is the most advanced among all formal verification techniques. It has the ability to decompose a large problem into a set of smaller ones. The model checker is widely used in protocol verification, whereas the equivalence checker and theorem prover have been actively used in microprocessor verification.

A typical SoC design contains a microprocessor, a bus protocol, various IP blocks, memory blocks, and a software component. Ideally, these components are verified using one formal verification technology. In most cases, this technology may not be the most suitable technology to be used on all types of components. Verifying SoC design necessitates the use of various formal verification technologies. There is a challenge to find a formal tool which offers such technologies. On the other hand, most formal verification tools were created based on specific technologies which differ from each other and make such formal verification tools unique. In order that more than one formal tool can be used, integration problems must be resolved.

In this dissertation, we explore the utilisation of formal verification to conduct SoC design verification. The approach will provide the answer to the following problems:

- How to represent each component of the system.
- What formalism is suitable for each component.
- How to integrate the components.
- How to verify the system.

1.3 The Approach of this Research

The SoC design methodology relies on using IP blocks or VCs to create a more complex system. Such reuse is the primary key in reducing the time to market for new products. The idea to reuse existing formal proof techniques in the system level verification is proposed, that builds a formal system similar to the SoC design. The formal system is called a *formal verification platform.* In contrast with SoC design which normally is implemented in a standard *Hardware Description Language* (HDL) such as *Verilog HDL* or *Very High Speed*

Integrated Circuit HDL (VHDL), the formal models are constructed using various formalisms. These formalisms depend on the tool used in the verification of each model. Finding a formal tool platform which offers the relevant formal technologies will be part of the challenge.

All formal verification tools come with their own limitations. On the one hand, the automation offered by equivalence checking and model checking make these two technologies popular among verification engineers. On the other hand, the automation comes with a capacity limitation which limits the tools to deal only with a small circuit size. While the problem decomposition from a theorem prover is capable of handling large and complex systems, it requires substantial involvement from the user to operate the tool. Significant effort has been expended to combine various tools, from the pragmatic semi-formal verification method that combines the conventional and formal verification methodology to the mixed verification tools which combine different verification techniques [28]. The extended system is delivered either by extending an existing tool with a new technology or by integrating various technologies in an integrated environment [2, 3, 16, 26, 31, 55]. In this dissertation we present one architecture which has a complete set of formal verification technologies in a single environment.

The objective of this dissertation is:
to demonstrate the feasibility of defining a heterogenous formal tools system which can be applied in the system level verification of SoC designs. The approach uses a formal verification platform and allows the reuse of existing formal proofs from the individual components.

1.4 Contributions

The contributions of this dissertation are as follows:

- A heterogenous formal verification environment is defined. It is an environment where a selection of formal tools are integrated to offer a complete set of formal verification techniques. The combined capabilities of this environment provide verification engineers the flexibility to choose the most suitable formalism for each component of SoC design.
- A formal verification platform is proposed to perform formal validation on SoC design. A formal verification platform is a platform where a verification engineer can easily integrate various formal models in a single formal verification environment.

- A methodology to obtain the generic properties of a formal model is developed. The properties can be reused in higher level validation such as in the interface level and system level.
- The feasibility of the methodology is demonstrated in two case studies.
  - The verification of SIP.
  - The analysis of the ARM platform.

In the first case study, we demonstrate our methodology in verifying system level properties of SIP. SIP contains two master modules and one slave module. Inter module communications are managed by an arbiter which is specifically built for the system. Using formal verification, it will be shown that the system has liveness properties. The correct execution of the software component embedded in the system's memory will also be verified. A report of this work was published and presented in [77].

In the second case study, the methodology to design an application using a generic or standard platform will be shown. The required specifications for each module are obtained by analysing the application's requirements along with the standard platform.

The standard platform contains an ARM7 processor and an AMBA bus protocol. A formal model of the ARM7 processor in the LISP programming language was developed. In this way, the processor's formal model can be analysed using symbolic simulation by theorem proving as well as traditional simulation by execution. The AMBA-AHB bus protocol developed by the *Institute for System Level Integration* (ISLI) was used and was implemented in Verilog HDL. A set of generic bus properties was then developed by defining its input and output relations. This work has been accepted for publication in [78].

This dissertation is the first to propose a formal verification platform using an integrated heterogenous formal tool system to perform a system level verification for SoC design. Apart from a few scattered results [18, 19, 20, 52, 66], no other work has been done on this topic.

1.5 Outline of the Dissertation

The dissertation is organised as follows:

In chapter 2, a brief summary of the application of formal verification to digital hardware circuits will be discussed.

In chapter 3, a brief introduction to SoC will be presented, describing various aspects of SoC design, the technologies that drive the revolution in ASIC design, the differences between ASIC design and SoC design, the advantages and disadvantages of SoC design, the challenges and opportunity offered by SoC design, and the technologies that define SoC design methodology.

In chapter 4, the SoC design methodology will be described using an integration platform. The concept behind a formal verification platform and a formal verification environment will be explained, providing the justification of the selection of formal tools and a brief description of the integrated formal tools environment.

In chapter 5, a brief introduction to formal tools used in the formal verification environment will be presented and descriptions will be given for the physical connections that integrate those tools in such a way that information can be passed amongst them.

In chapter 6, the architectural description of SIP platform will be given, presenting the formalism of each SIP module and showing how these models are integrated to create an SIP formal verification platform.

In Chapter 7, the formal verification of SIP will be presented. The verification is divided into two parts. The first is the verification of liveness properties of the system. The second is the verification of correct execution of the software component which is embedded in the platform.

In chapter 8, the architecture of an ARM platform, a standard integration platform architecture based on the ARM processor and the ARM AMBA bus protocol will be described and the formalisation of the ARM7 processor and AMBA-AHB bus protocol will be presented.

In chapter 9, the development of reusable formal properties of the AMBA-AHB bus protocol will be described. The way protocol properties and processor properties are used to create an application and to develop the required specifications of the remaining modules which can satisfy applications' requirements will be shown.

In chapter 10, a summary of this dissertation will be presented along with discussion of future work.

Chapter 2

Related Work

Almost all formal hardware verification activities have been targeted to verify complex logic circuits, microprocessors, and bus-protocols. Most of the existing reported work in formal hardware verification for SoC design has been concentrated in the verification of the protocol or the interface [18, 66]. The effort closest to the work described in this dissertation is the work by Liao and Hsiung [52], and Choi et.al. [19, 20].

The goal of this chapter is to give an idea of which technologies and what approaches are used in the verification of the common components of SoC designs. The brief summary of microprocessor and bus protocol verifications will be presented in Section 2.1 and Section 2.2 respectively. A brief summary of the work done by Liao and Hsiung, and Choi et.al. will be presented in Section 2.3. A summary will conclude this chapter.

2.1 Formal Verification of Microprocessor Designs

Microprocessor design verification is an active area for formal hardware verification both in academia and in industry. Various formal techniques are used in this area. A selection of results in microprocessor verification are presented in this section.

The early efforts in microprocessor verification were concentrated in the full verification of non-pipelined microprocessor designs. Some notable results by Hunt, Joyce, and Birtwistle are presented in this section.

Hunt was the first to consider the problem of reasoning about the implementation of a handshaking protocol to synchronise data exchange between a microprocessor and external memory [84]. He implemented this microprocessor (FM8501) in LISP and verified it using the Boyer-Moore theorem prover. Hunt used an oracle to guess the length of wait state for the hand shaking operation. He proved the correctness of the system by showing that the microprocessor is correct for all possible oracles. This work was later extended to the verification of the FM9001 microprocessor [85]. FM9001 is specified at 4 levels of abstraction. The highest level layer is the instruction set specification. The lowest level layer is the actual hardware implementation. The verification is performed by proving that each layer is a correct implementation of the layer above it. By doing so, Hunt proved that the actual implementation of the microprocessor implements the instruction set specification. This verification is also performed using the Boyer Moore theorem prover. Both FM8501 and FM9001 are non pipelined microprocessors.

Joyce designed the TAMARACK-3 microprocessor and used the HOL theorem prover to verify the design [46]. TAMARACK-3 is a non pipelined microprocessor and has an input for interrupt request. It can be interfaced to external memory and can be operated in fully synchronous and asynchronous mode. Joyce verified the correctness of the microprocessor operations in both synchronous and asynchronous mode.

The final example is the verification of the SECD microprocessor by Birtwistle and Brian using the HOL theorem prover [12, 35]. SECD is a moderate complexity microprocessor. It is modelled in two levels of formal specification, the top level specification and the low level RTL implementation. The correctness of the microprocessor is achieved by showing that the sequence of operations in the RTL level correlates with the specification transition at the more abstract top level.

Recent microprocessors have become more complex with the inclusion of features such as out-of-order execution, complex multi-stage pipelines, and co-processors. Some of the notable results by Sawada, Srivas, Greve, Patankar, and Fox are presented below.

Sawada used the ACL2 theorem prover to verify an out-of-order pipelined processor [68]. The processor includes out-of-order execution and speculative instruction fetch. He proposed the use of a table called *Micro-architecture Execution Trace Table* (MAETT) to store an execution trace of instructions representing states in the implementation. MAETT captures the past history of the processor's execution process and helps to easily define various pipeline properties.

Srivas and Miller used the PVS theorem prover to verify the AAMP5 microprocessor [75]. The AAMP5 has a large and complex instruction set, multiple addressing modes, and a pipelined implementation. The verification problems were decomposed into two sub-goals: a part that reasons about stalling behaviour and a part that reasons about individual instructions in the absence of stalling.

Greve used the symbolic simulation technique in the PVS theorem prover to verify the JEM1 microprocessor [37]. JEM1 is the first direct execution Java processor. The instruction set of JEM1 is a superset of the *Java Virtual Machine* (JVM). Symbolic simulation in PVS is still in its early stages. As a result, the verification suffers from slow symbolic simulation speed.

Patankar used the BDDs-based *Symbolic Trajectory Evaluation* (STE) technique to verify the ARM6 microprocessor [61]. The verification is conducted using the Symbolic Trajectory Evaluator with trajectory assertions at the gate-level implementation of the microprocessor. The instruction set architecture is specified as a set of abstract assertions. These abstract specifications are used to generate a set of trajectory assertions. Although the verification is completed only on the bitwise-OR instruction, it can be extended to other instructions. The only exception is that with the current model, the verification of multiplication instructions may yield an exponential memory complexity.

Fox also verified the ARM6 microprocessor using the HOL theorem prover [30, 31, 32]. In contrast to Patankar, Fox completely verified the abstract model of the ARM6 microprocessor. The microprocessor is specified in such a way that it can be executed either symbolically or with variable free (constant) terms. Similar to Greve, the execution of the microprocessor model is slow, especially when it is compared to the symbolic simulation speed of ACL2.

The complexity of microprocessor circuits leads verification engineers to concentrate their efforts to guarantee the specified performance of a module before it is integrated into a chip. Industrial communities have used formal techniques in the verification of complex arithmetic functions. Some notable results are by the team at Intel and AMD.

The Intel team verified the *Floating Point Adder* (FADD) circuit of the Pentium® Pro microprocessor [1] using an in-house tool called Forte. Forte is a verification environment which integrates model checking engines, BDDs, circuit manipulation functions, theorem proving, and a functional programming language. The correctness of the FADD circuit is shown by decomposing the goal into subgoals based on the type of operation (addition/subtraction) and differences between two components. Every subgoal is verified using the model checker. The completeness coverage of problem decomposition is ensured by maintaining goal decompositions in the theorem proving environment.

Russinoff verified the FADD circuit of AMD Athlon^{TM} microprocessor [67]. He used the ACL2 theorem prover as the verification environment. The verification approach is by using two set of descriptions of the circuits, the actual RTL code and a simplified version. These circuits were translated into the ACL2 logic as

ACL2 functions. The correctness of the circuit was obtained by proving the equivalence of these two functions.

At Intel, Harrison verified the trigonometric functions of INTEL's IA-32 [39]. He verified the circuit using the HOL light theorem prover. The development of proofs was very tedious. The tool allow users in developing program to automate the proof process.

2.2 Formal Verification of Bus Protocol Designs

Another active area in formal hardware verification is protocol verification. Various specifications, such as cache coherence protocols, PCI bus, and FireWire bus, have been verified using various formal techniques. The automation of model checking technology make model checker tools the preferred option to be used in bus protocol verification.

Dill et.al. showed that an automatic protocol verification system can be used very early in the design phase to catch design errors. They used the Murφ system to verify a cache coherence protocol system [29]. One of the techniques used in the verification was *down scaling*, reducing the complexity of the system by pretending the system was small. The idea behind down scaling is that most of the bugs in the protocol can be demonstrated using two or three processes. Although this technique did not guarantee the total correctness of the system, it gave better coverage in detecting errors than standard simulation techniques.

Clarke et.al. verified the IEEE FutureBus$^+$ cache coherence protocol using the SMV model checker [22]. The protocol is a standard to define a hierarchical bus structure system for many processors and caches. In this work, Clarke verified a structure with three bus segments, eight processors and 10^{30} states. The verification can be extended to cover as many system configurations as possible by using induction in the model checking.

Although automated model checking is the preferred technology in bus protocol verification, Mokkadem et.al. have shown that a bus protocol system can also be verified using a theorem prover. They show this by verifying the PCI 2.1 bus transaction ordering problem using the PVS theorem prover [59]. The proof result was generic; it covered any acyclic network of a finite number of PCI buses. In the verification, induction was used extensively.

Calder and Miller analysed the tree identification phase of the IEEE 1394 FireWire bus protocol system using the SPIN model checker [15]. They overcome the model checking limitation of having to have a fixed configuration for each process by using a Perl script to generate the model and subsequent model checking runs. The verification coverage is then checked manually.

2.3 Formal Verification of SoC Designs

2.3.1 Timed Automata Formal Verification Platform

Liao and Hsiung proposed a *Formal Verification Platform* which is similar to the proposed idea in this dissertation [52]. They defined verification flows for the Formal Verification Platform which is similar to the standard ASIC design and validation flows. The platform consists of three components: the *SoC environment configurator*, the *SoC system integrator*, and the *State Graph Manipulators* (SGM).

Initially, a set of IPs are modelled as *Timed Automata* (TA) models. TA is a modelling technique which describes the behaviour of a component in a real time concurrent system. In this approach, it is possible to define a multiple clock system. The SoC environment configurator is then used to set up an SoC environment. In this phase, users select all components which will be used to create the SoC design. The next stage is the integration of the IP models specified in the SoC environment configurator. The components are connected through shared variables and synchronisation labels. Shared variables contain register variables which provide numeric information about the system and clock variables which provide timing information. The synchronisation labels model the synchronous behaviour of the timed automata. Finally, the SGM kernel is used in the verification of the system using a model checking technique. Properties used in the verification were specified in *Computational Tree Logic* (CTL).

2.3.2 Model Checking a System on Chip Design

The SoC development group at Samsung Electronics have reported their experience and methodology used in the verification of S3C2400X in [19, 20]. The S3C2400X is an embedded SoC product which is composed of an ARM920T processor core and 16 IP modules, such as *Universal Serial Bus Host* (UHOST), *Direct Memory Access* (DMA) controller, and etc. The AMBA bus protocol is used to manage the communication between these modules. All verifications were conducted using the SMV model checker.

The IP modules used in S3C2400X are classified into three equivalence classes: old IPs, newly designed IPs, and newly bought IPs. An old IP is an IP which has been used in a previous product and has been verified in silicon. In most cases, the interface logic of this IP is re-designed to match the requirement of AMBA bus protocol. Formal verification is applied to verify the interface logic of this IP. A newly designed IP is an IP which is designed in-house. The correctness of this IP is obtained by verifying it using formal verification. A newly bought IP is an IP purchased from a third party vendor. Verification will be performed if the module needs to be modified. For example, UHOST in S3C2400X is a newly bought IP. A wrapper is developed to bridge the differences between the UHOST *Host Controller Interface* (HCI) bus interface and the AMBA bus interface. Formal verification is used to verify the wrapper. A model of the HCI bus interface is also produced for the verification.

In this work, a golden model of a module is produced through the following design and verification processes. First, a module is designed either in Verilog or SMVL. If the module is implemented in Verilog, it is transformed into SMVL. The expected properties of the module are specified in CTL. The module written in SMVL and the CTL properties are model checked. If the verification succeeds for all properties, the SMVL module becomes the golden model. This golden model is then translated into Verilog and becomes the Verilog golden model. If for refinement purposes the Verilog code is changed, the module is re-verified using an equivalence checker or a model checker.

The SMV model checker is based on *Binary Decision Diagrams* (BDDs). It searches every possible state that the system being verified can reach. SMV used to suffer from state space explosion. To avoid this limitation, two basic strategies are used in the verification: data abstraction and case splitting. Data abstraction is applied to the data bus and address bus to avoid the explosion of the BDD's size. Case Splitting is used to decompose a goal into several subgoals which make the model checking easier and more efficient. Finally, all CTL properties used in the verification are manually checked against the RTL for coverage analysis.

2.4 Summary

This chapter provided a summary of the existing results in processor and bus protocol verification. It also summarised one industrial case study of SoC verification. In general, any formal verification technology can be applied to verify digital hardware circuits. The only question is which one is the most suitable for the task.

Verification of microprocessor designs are dominated by theorem prover and symbolic simulator technologies. The common design practice of validation by simulation has inspired semi-formal verification methods that combine conventional and formal verification methodologies [28]. It also inspired some use of symbolic simulation techniques in a theorem prover environment. On the other hand, the limitation of a symbolic simulation system which can only handle a limited design size is overcome by adding on top of the system a layer of theorem proving [1, 40].

Bus protocol systems can be verified using a theorem prover or a model checker. The automation of model checking systems make them the preferred choice in the verification of bus protocols. On the other hand, model checking systems suffer from the limitation that they can only be used to verify a fixed configuration model or process. There is a trend to extend the model checker with a layer of theorem proving capability which offers induction schemes and automation of the management of problem decompositions [24, 50, 55].

Chapter 3

An Introduction to System On Chip

In this chapter, a brief introduction to *System on Chip* (SoC) technology is provided. The chapter is divided into six sections. A brief history of ASIC technology and the technology behind it is presented in Section 3.1. In Section 3.2, an overview of SoC technology is presented. First, the difference between ASIC and SoC designs will be shown. Then, the advantages and disadvantages of an SoC design will be explained. Finally, the issues in creating an SoC design and the challenge in validating the design will be discussed. The evolution of ASIC design methodology is presented in Section 3.3, describing the technologies that drives the revolution of design methodology. In Section 3.4, the methodology behind reusable design will be described by exploring different forms of reusable design and the correlation between design reuse and design methodologies. The issue of validation is presented in Section 3.5. Finally, a summary of this chapter is presented in Section 3.6.

3.1 Fundamentals of ASIC Technology

The micro-electronics era was started in 1958 when Jack Kilby came up with a new idea for creating a more reliable circuit design. His suggestion was to create the design by integrating a number of transistors in a single package, instead of making it from stitching a lot of components together. Then the invention was known as the *solid circuit*. Today, it is known as the *integrated circuit* (IC) or the *silicon chip*. One year later, Robert Noyce developed a new concept of miniaturisation, creating transistors on the surface of silicon wafers. This invention enabled more transistors to be integrated in a tiny space and produced silicon chips in mass production. Since then, the manufacturing technology has moved forward at a tremendous speed [21].

In 1965, Gordon Moore made an observation on the growth of manufacturing technology. He made a very famous prediction, that the number of transistors on a given piece of silicon will be doubled every couple of years. This is known as *'Moore's Law'* [43]. Moore's prediction describes an exponential growth in transistor density, the result of which was that more complex designs could be integrated in the die. In addition, the shrinking size of transistors makes ICs cheaper, more powerful, and more plentiful.

One way to get some understanding of the impact is by examining the changes in a single *metal oxide semiconductor (MOS) transistor*. Consider Figure 3.1(a) which illustrates a MOS transistor. It consists of a channel with source and drain at each end, and a thin layer of oxide with a gate layer on top. The length of the channel is denoted by L. The width of the wiring is indicated by W. The time to activate the transistor is affected by the thickness of the oxide and is defined as t. When the manufacturing technology advances by a size reduction of α, the length of the channel is reduced from L to L/α. The width of the wiring is also reduced from W to W/α. As the result, the transistor size is reduced by α^2 and the transistors density of the die is increased by α^2. The reduction of the oxide gap reduces the transistor activation time from t to t/α. Now, the transistor can be activated with a speed of t/α. The reduced size of the MOS transistor by α is shown in Figure 3.1(b).

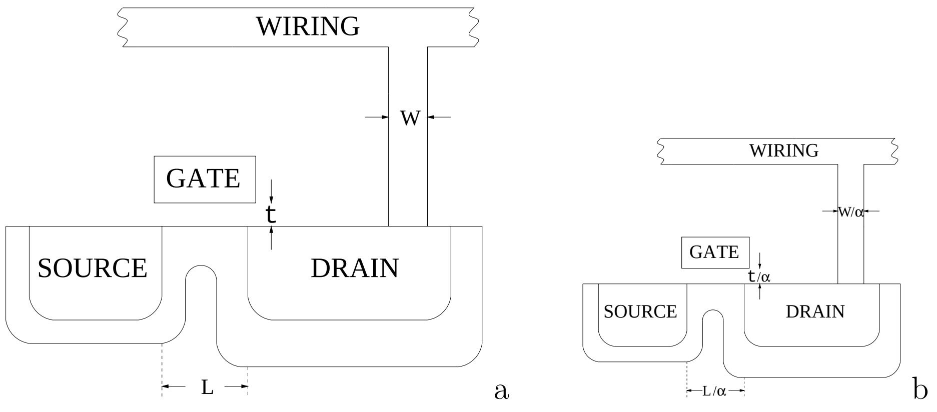

Fig. 3.1: (a) A MOS transistor original size (b) A MOS transistor reduced size by α

Over more than four decades the technology has advanced as predicted by Gordon Moore. Since then, the IC's design complexity has moved forward from SSI (Small Scale Integration), MSI (Medium Scale Integration), LSI (large Scale Integration), VLSI (Very Large Scale Integration), ULSI (Ultra Large Scale Integration), and recently, GSI (Giant Scale Integration) [63]. The term VLSI also used to refer to ULSI and GSI design. Today, we have technology which can create a system containing a billion transistors on a single piece of silicon. Chip designers can put more functionality on a single chip. At the same time, a single chip system provides the high performance engine which is needed by complex applications.

This chapter describes the basic idea of SoC, the components that support the creation of a SoC system, and the challenges that SoC designers have to face.

3.2 System on Chip Overview

The reminder of this chapter is based on the ISLI course material on SLI Methodology Overview [42].

3.2.1 ASIC vs SoC

The practice of integrating system functions on a single platform has been around since the beginning of microelectronics. In the early days, systems were built by connecting standard ICs such as TTL ICs, microprocessor chips and memory devices, on a single board or multiple boards (Figure 3.2). *Printed Circuit Boards* (PCBs) are the platform used to integrate the components. As the technology advanced, the integration of standard ICs to perform specific functions or a set of functions shifted the integration from board level to chip level. Larger components such as microprocessors and memory devices are still unpractical to be integrated together. The integration of these ICs in the system largely remains at the board/PCB level.

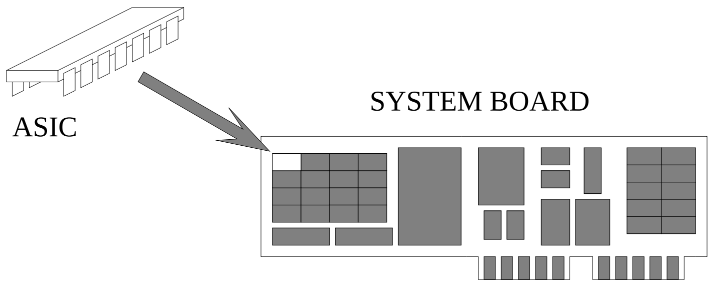

Fig. 3.2: system on a PCB builds from ICs/ASICs

As the technology continues to improve, more and more functionality can be put into a single chip. The latest technology has again shifted the design practice from *Application Specific Integrated Circuit* (ASIC) design towards the *System on Chip* (SoC) design. Today's SoC chip integrates components of a system on a single silicon substrate (Figure 3.3). SoC is defined as a single complex IC that is designed by integrating together multiple stand-alone VLSI designs, known as *Virtual Components* (VCs), to produce full functionality of a complete end-product for an application. The SoC design methodology offers the ability to integrate various components in a plug-and-play approach. The components include programmable processors, on-chip memory devices, hardware to perform specific functions, and peripheral devices. It also includes the possibility to integrate software components along with the hardware parts.

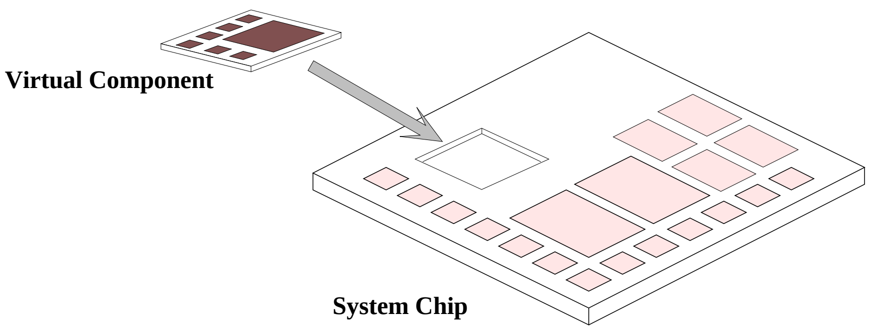

Fig. 3.3: System on a chip builds from VCs/IPs

3.2.2 Advantages and Disadvantages of SoC Design

SoC technology offers advantages in various aspects. It increases the reliability and manufacturability of the design. A complex SoC design is created in a single die, manufactured using common substrates and materials. This allows for easier manufacturing of the design. At the same time, all modules and their interconnections are created together. This reduces a source of unreliability which can occur in the system. The integration reduces the possibility of failure in the packaging and pins internal connection. It also reduces the failure of the solder joints, wire joints, and PCB tracks.

The technology also delivers an increase in design performance. In SoC design, modules' interconnection is moved from slow off-chip communications to fast on-chip communications. As a result, the communication speed between subsystem modules is increased. Integrated modules reduce the off-chip capacitance effect between module interconnections. The total die size for the complete application is reduced when multiple functional blocks are packed together in a single chip. The expensive I/O pads from every module, when manufactured individually, are replaced by a combined and more efficient single chip I/O pad. SoC has simplified packaging as the system may have fewer external connections and interconnections between its subsystems. The power consumption is also reduced, as the power consumption per transistor is falling and power dissipation is reduced. This will allow SoC design to be used in low power applications which can run on batteries. Furthermore, each generation of technology brings down the transistor price. This allows for low cost SoC chips.

SoC technology enables designers to be more creative and explore the creation of more advanced architectures. For instance: designers can design an integrated multi-processor system in a single chip, they can explore the use of distributed memory systems and parallel processing systems which manage task sharing between processors, and they can also experiment with various in-chip communication architectures such as network on chip and asynchronous communication systems.

On the down-side, the technology also comes with issues that need to be dealt with. Extra masking layers in manufacturing are needed to combine standard logic/digital components and other components (such as SRAM, DRAM, flash memory, analog/mixed signal). This is due to the incompatibility of manufacturing technology for the components. As a result, it will increase the complexity of the manufacturing process and raise the cost of production.

The single package system raises the issue of power dissipation. It also raises the complexity of dealing with power distribution to the subsystems and the peak current demands at clock transitions. Signal integrity will be a major concern in SoC design. The issue of digital noise injection which is fed into sensitive analogue components appears because a common substrate is used to manufacture the design. Digital noise is responsible for the increase of random switching in low voltage systems. Although the size reduction of digital components leads to a lower voltage system, designing analog circuits at low voltage is not easy. Furthermore, external drive interoperability requires a high voltage system, raising the issue of voltage restrictions in the design.

SoC design relies on using existing *Intellectual Property* (IP) blocks. Acquiring third party IP usually requires a considerable amount of time to find the right IP and negotiate the terms of agreement. This will raise the design time and cost which affects the up-front non-recurring engineering (NRE) costs.

Designing using IP components raises issues in validation and testability. How does the designer know if the IPs work when they are acquired? How does the designer know the design is going to work before building it? The SoC approach forces the designer to start thinking about a test strategy, prior to the commencement of the design phase. The validation strategies will need to test subsystems and the system as a whole.

3.2.3 SoC Issues and Challenges

SoC technology provides an integrated solution to system design problems. Success relies on how to use the appropriate design approaches. The challenge is in finding the right one. In general, we group these dilemmas into three categories [11]:

- How can the designer create an SoC system?
- Where can they get the components?

- How can they build the system?

The first point arises when the designers are pressured to produce the product with a shorter time to market, while the complexity of the system increases. One way to deal with this issue is to shorten the design cycle by reusing as many existing designs as possible. In SoC design, these components are often called *virtual components* (VCs) or *intellectual properties* (IPs).

The second point addresses the problem in creating or acquiring reusable components. In the past, most system developers created a system by combining various components (ICs) on PCBs. In most cases, the components come from various vendors or sources which specialised in specific areas or expertise. Similar to this, SoC design is created by integrating various VCs. If the design-house has the expertise, they can create their own VCs. If not, then they must rely on acquiring third party VCs, which come from the effort of IC suppliers in transforming existing design into VCs.

The final point deals with the approach needed to build the system. One approach is to create the system by mixing and matching reusable components sourced from internally and externally developed VCs. The mixing and matching approach offers the ability to connect and integrate VCs from different sources. But it does not guarantee that the system will actually work. This practice brings the following challenges to integrating SoC design:

- The challenges in the interoperability of VCs.
- The integration of VCs in the context of system-level design.
- The interoperability of Electronics Design Automation (EDA) tools.
- The validation of the design.

In a board design, designers are rarely concerned with the technology to create and test the components. They have accepted that the components match the given specification. They only need to understand the component interfaces and their operational models. When a problem rises, they fix the system by adding some glue logic or board reworks. In SoC design, the components come as VCs. In contrast with the board design where every component has been standardised, VCs can appear at various levels of abstraction. They can come in the form of soft-cores, firm-cores, and hard-cores. A detailed description of these cores is presented in Section 3.4.1. Most likely, every VC supplier creates and verifies its VCs. There is no common interface between VCs.

There are several obstacles that need to be considered in integrating VCs. In practice, SoC designers have to integrate system level representation of various

VC blocks. Often the representations of these blocks are defined in different languages and environments. A conversion is needed for every representation to fit the system integration environment. Unfortunately, there are few tools which can deal with these issues. The use of a processor also raises another issue. Typically, a processor is always associated with a set of proprietary systems (such as busses, memory, and peripherals) and tools (such as assemblers, compilers, linkers, drivers). Furthermore, the problem is compounded by the unlikelihood of existing software applications being compatible between different processor systems.

Almost all ASIC companies use various EDA tools in developing their designs. In most cases, the tools will not come from a single vendor. Every EDA vendor develops their own proprietary data format for their file standards, such as report files, library files, etc. These data format differences force the design company to develop translators and filter programs to bridge and stitch the tools and create a suite of design tools. This set of tools define the company's design flow or design methodology. Their biggest shortfall as a result of this situation is that they have to keep pace with tool vendors in updating the programs every time tool vendors upgrade their tools or replace the tools with new ones. The situation becomes worse when the ASIC company has to use third party VCs, which may be designed using a different tool suite. Standardisation will be the key factor in tool inter-operability.

The biggest challenge in SoC design is in validating and testing the system. Validating and testing SoC designs differ from board design. On a PCB, the system can be probed through the components' physical pins. In an SoC, all components (VCs) are internally connected which makes probing the connections extremely difficult. One way to overcome this hurdle is to make the system self-testable by using *Build-In Self-Test* (BIST) as part of the system, or by implementing a *partial-scan* or even a *full-scan* test. The decision to add self-test functions has to be made as early as possible in the design process, as it will be hard to extend a nearly built system. Third party VCs may come with no self-test scheme, or they may come with different self-test schemes. SoC design must be able to adapt to any situation to ensure the design can be quickly tested.

3.3 Towards SoC Design Methodology

Every advancement in microelectronics processing technology is always followed by the development of new design technology [17]. This new design technology, a so-called *linchpin technology*, becomes the building block to lead the design entering the next generation of design methodology. The design methodology responds with an adaptation to the new design process resulting in an incremental increase in productivity. It alters the relationship between the designers and the design by introducing a new level of abstraction.

Over the past few decades, several linchpin technologies have been invented. For example, the invention of gate level simulation enabled an increase in design validation capacity. The effectiveness of a simulator is defined by the accuracy of the gate level models and their associated libraries. The inclusion of a gate level simulator in the design process means the designer accepted the limitation and boundary brought by the simulator. As another example, consider *Register Transfer Level* (RTL) synthesis technology that provides another path to increase designer productivity. It requires the transition from gate level design to the RTL-based design and validation methodology. The effectiveness of synthesis technology is defined by the predictability limitation of the optimisation technology. These technologies have led to fundamental changes in the design methodology.

A linchpin technology always comes along with its specific design methodologies. In general, these design methodologies can be grouped into four groups:

- *Area-Driven Design* (ADD)
- *Timing-Driven Design* (TDD)
- *Block-Based Design* (BBD)
- *Platform-Based Design* (PBD)

These design methodologies are differentiated by their design capacity, level of reusable block, and the design technology used in creating the design.

3.3.1 Area-Driven Design

Area-Driven Design (ADD) is the most basic and the simplest methodology used in creating ASIC designs. It is driven to achieve the primary goal target in creating a design which can fit into a limited budget area. The designer is challenged to implement as much functionality as possible in a single piece of silicon. The ADD methodology is used to achieve small sized ASICs. Most ADDs are created from scratch and do not offer any design reuse.

The main ADD activity is in logic minimisation. The synthesis optimisation is to produce the smallest design which can meet the intended functionality. In this methodology, no floor planning information is used at the RTL or gate level analysis.

ADD can be identified by two linchpin technologies: RTL/logic synthesis tools and gate-level simulators. RTL synthesis increases the design productivity from the traditional schematic design. It helps the designer to identify the best implementation that meets the area goal. A gate-level simulator enables the designer to quickly validate the design and identify any incorrect functionality before and after the synthesis.

3.3.2 Timing-Driven Design

Timing-Driven Design (TDD) is a methodology for optimising a design in a top down, timing convergent manner. It is driven by the design requirement for meeting performance or power consumption. The methodology is used to achieve a moderately sized complex ASIC design. In general, the complexity of a TDD circuit is between 5000 to 250K gates. It is primarily a custom logic design, offering a very slim possibility of design reuse.

The TDD methodology imposes a more floor plan-centric design methodology that supports incremental changes of the design. The floor planning and timing analysis tools can be used to determine the location of placement sensitive areas, allowing the results to be tightly coupled into the design optimisation process.

TDD relies on three linchpin technologies: interactive Floor-Planning (FP) tools, Static Timing Analysis (STA) tools, and using compilers to move design to higher abstraction with timing predictability. FP tools give accurate estimation on the delay and area early in the design process. They address the timing and area convergence problems which occur in the design process between synthesis and 'place and route'. STA enables a designer to identify timing problems and perform timing optimisations across the entire ASIC. It reduces the validation stress in catching bugs using a slower timing-accurate gate-level simulation. The advancement in compiler technology enables the designer to move the design into higher abstractions while retaining timing predictability. For example, behavioural synthesis provides an operational vehicle for planning and implementing data-path dominated design rapidly. A higher-level abstraction will move the critical decision trade-offs into the behaviour level, which leads to an efficient data-path layout.

3.3.3 Block-Based Design

Block-Based Design (BBD) is the design methodology used to produce designs that are reliable, predictable, and can be implemented by top-down partitioning of the design into hierarchical blocks. It introduces the concept of creating a system by integrating blocks of pre-designed system functions into a more complex one. The methodology is used to create medium-sized complex ASICs with complexity between 150K to 1.5M gates. BBDs are primarily created as custom logic designs. In comparison to TDD; BBD offers a better chance for reuse, although in reality, very few BBDs are reuseable.

Initially, the systems are behaviourally modelled before the process of design implementation starts. The models are analysed for the trade-off of hardware/software partitioning and functional hardware/software co-validation. The partitioned components are mapped onto specified functional RTL blocks. The blocks are designed to meet the budgeted area, timing, and power constraints. The top-down planning creates individual blocks to allow synthesis to analyse timing hierarchically. Then, the designers can choose to perform a flattened or hierarchical analysis on the final routing.

BBD relies on three linchpin technologies: application-specific high-level system algorithmic analysis tools, block floor planning, and integrated synthesis and physical design. The analysis tools provide the environment for modelling a system's algorithm and environment. They can be linked to HDL validation tools through co-simulation technologies and to HDL-based design capabilities, such as RTL synthesis via HDL generation and behavioural synthesis. Block floor planning facilitates interconnected management decision-making based upon RTL estimations through faster area, timing, and power convergence. It provides a specific constraint budget for each component/block in the context of the top-level chip interconnection. This will enable the designer to focus the optimisation for each individual block. Integrated synthesis and physical design enables a designer to manage the increased influence of physical design effects during the synthesis process by eliminating the need to iterate between separate synthesis, placement and routing tools to achieve design convergence. In the integrated system, the synthesis can meet the top-level constraints in a more predictable manner.

3.3.4 Platform-Based Design

Platform-Based Design (PBD) is a methodology which is driven to increase productivity and time to market by extensively using design reuse and design hierarchy. It expands the opportunities to speed-up the delivery of derivative products. PBD achieves high productivity through extensive and planned design reuse. Productivity is increased by using predictable, pre-validated blocks that have standardised interfaces. The methodology focuses on better planning for design reuse and less modification on the existing functional blocks. PBD is used to design large sized complex ASICs with design complexities greater than 300K gates.

The PBD methodology separates the design into two categories of activity: block authoring and block integration. Block authoring uses a methodology which is suited to the block type such as TDD or BBD. Blocks are created with standardised interfaces so they can be easily integrated with multiple target designs. Block integration focuses on designing and verifying the architecture of the system and the interfaces between the blocks.

PBD focuses around a standardised bus architecture and increases its productivity by minimising the amount of custom interface design or modification of the blocks. The test for the design is incorporated into the standard interfaces to support each block's specific test methodology. This allows for a hierarchical, heterogenous test architecture.

PBD relies on three linchpin technologies: architectural design tools, physical layout tools, and VC authoring validation tools. The architectural design tools provide the environment to do high-level and system-level design. They also provide the vehicle to perform partitioning of hardware and software and perform functional validation by leveraging comprehensive models of VCs. The physical layout tools focus on bus planning and block integration enabling predictable, constraint-driven, hierarchical place and route of PBD designs. PBD validation focuses on the interfaces: block to block, block to bus, hardware to software, digital to analog, and chip to environment. The VC authoring validation tools provide a thorough validation which can be used throughout the validation levels.

3.4 Design Reuse

Design for reuse is a methodology in creating a design that can be reused. A reusable design implies it is a correct and robust design. It comes with good documentation, good code implementation, well designed validation environments, and robust scripts for synthesis and validation.

In general, the designs (IP cores) have to follow some guidelines to be fully reusable. First, a reusable core is created to solve a general problem. The core must be easily configured to fit different applications. Second, the core is designed for use in multiple technologies. The core must have an effective porting strategy for mapping onto new technologies. If the core needs to be re-synthesised, the synthesis scripts have to produce satisfactory result quality with a variety of libraries. Third, the core must be designed for simulation with a variety of simulators. This implies each IP must come with VHDL and Verilog designs. The validation test-bench of each version also needs to be available and should work with all major commercial simulators. Fourth, the core must be verified independently of the chip in which it will be used. Reusable designs must have full test-bench and validation suites that afford very high levels of test coverage. Fifth, the core must be verified to a high degree of confidence. This means a rigorous validation as well as building a physical prototype that is tested in an actual system running real software. Finally, it must be accompanied by full documentation in terms of appropriate applications and restrictions. Core configuration restrictions and parameter values must be clearly stated. Interfacing requirements and restrictions on how the core can be used must also be documented.

3.4.1 IP Cores

By definition, IP cores are pre-designed and pre-verified complex functional blocks. According to their properties, IP cores can be distinguished into three types of cores: soft-cores, firm-cores, and hard-cores.

Soft-cores are architectural modules which are synthesisable. They offer the highest degree of modification flexibility. On the other hand, a lot of physical design issues need to be faced before the core can be fabricated. This makes the soft-cores very unpredictable in terms of performance. A synthesisable soft-core consists of a set of technology-independent HDL files, synthesis constraints, test-bench and validation information and adequate information.

Firm-cores are encrypted black boxes that are integrated into design flow in the same way as library elements. Firm-cores are delivered as a mix of RTL code and a technology-dependent net-list, and are synthesised with the rest of ASIC logic. They come ready for routing analysis and do not present significant difficulties for floor-planning, placement, and routing. They have the same routability properties as soft-cores. The performance of the block is still unpredictable.

Hard-cores are mask and technology-dependent modules that already have physical layout information which give predictable performance. The key deliverable is a fully verified layout in *Graphical Design System II* (GDSII) format, along with a design for a test structure and test patterns. The drawback of hard-core is that the cores can not be customised for a particular design application. Hard-cores require more model support than firm-cores, which increases development cost. On the other hand, the usage cost is lower because timing validation, test strategies, etc., have already been built into the design. Monolithic hard-cores create a jigsaw puzzle problem for ASIC layouts. When more than one hard-core is used, the ordinary place and route techniques cannot be used due to the existence of a strange, non-rectangular area left for routing other non-core logic.

3.4.2 Models of Reuse

In the early days, design reuse was more opportunistic. As the technology to support design reuse matured, design reuse was better planned and considered in the early phase of the design process. Based on how design reuse models are developed, they can be grouped into four categories; personal, source, core, and VC. The evolution of design reuse is shown in Figure 3.4.

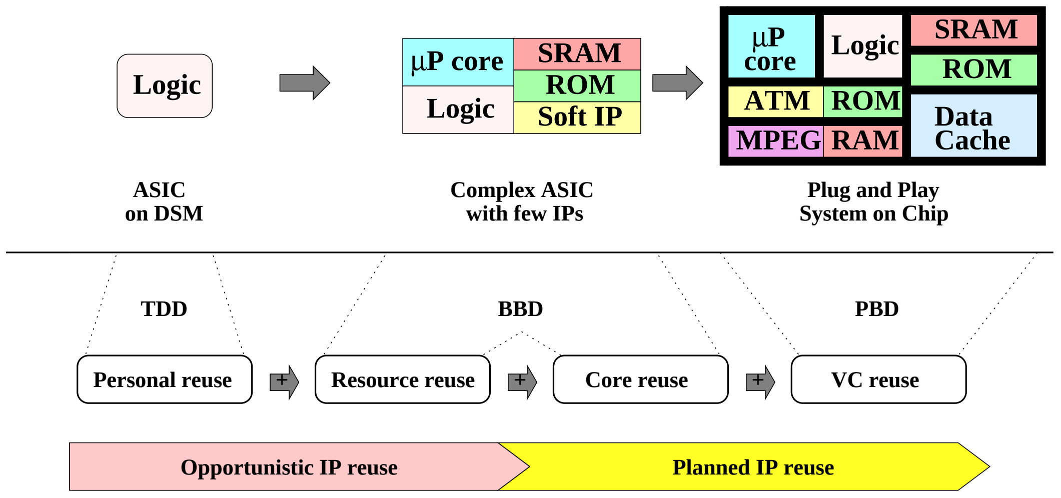

Fig. 3.4: Evolution of design reuse models

Personal design reuse appears in the TDD methodology. It is based on re-applying personal or team design experiences to produce derivative projects. The simplicity of the personal design reuse model make it only portable to single-thread operations. In most cases, the model is not scalable and depends on retraining key personnel who have acquired the knowledge.

As the design flow moved from TDD to BBD, the model became more reusable. It evolved from a personal design model to a source design model. In source design reuse, the source code of the design is provided either at the RTL-level or netlist-level. In this category a design can be modified to meet specific design requirements or constraints. On the other hand, if re-validation is required for every change made, then the reuse of this model will be less productive. The model also suffers from poor predictability of performance, area, and power consumption.

Along with the maturity of BBD methodology, the model of design reuse has also improved. More core design reuse models are available in firm and hard form. The information on timing, area, and power are sufficiently documented. Information exists on whether the core has been used and on the process of core integration. However, in this stage the core may still need to be modified to adapt to the bus architecture, power constraints, clock and test structures.

Finally, the transition of IP into VC creates components for plug and play environments. The VC model are pre-characterised, pre-verified, pre-modelled blocks which are targeted for a specific system integration environment and application domain.

3.5 Validation Issues

Validation is an integral part of the design process. It consumes 60 to 80 percent of overall design effort [10]. The complexity of SoC designs makes the validation task more important. It extends the traditional validation task from bug catching at block level to the system level with additional issues in block interfacing, and hardware/software integration.

In system-level validation, the system-level behaviour model is developed. A test-bench and test suites are developed to verify the model including test suites for software components. A validation strategy consists of at least three steps. The first step is the functional validation of each individual block as stand-alone units. The second is the validation of system interfaces between blocks at the transaction level and data transfer level. Finally, the full chip is validated to correctly execute the application software [49].

In block level validation, every module/block is rigorously validated. The validation includes compliance, corner case, and random testing. The compliance test verifies that the block complies with the standard published specifications, especially the standard interface specifications. Corner-case testing explores testing complex scenarios which are believed to be able to break the system. While in the first two tests, the scenarios used in the validation are normally anticipated scenarios, a random test uses a scenario which is not anticipated by the designer.

In interface validation, the interfaces between blocks are validated. The interfaces usually have a regular structure with an address-bus and a data-bus connecting the modules, utilising some form of control structure which manages the transaction process and data transfer between blocks. The activity in interface validation can be divided into two groups, transaction validation and data transfer validation. In transaction validation, all possible transactions for each interface are tested. A test-bench is created to cover the activity within the block and the transaction between blocks. If the overall system behaviour is correct, the chip is considered to be working correctly. After these validations are completed, it is necessary to verify that the blocks work correctly for any form of data and all sequences of data. A random data generator is used to create test cases. The test reveals that the block responds correctly to data sequences that the designer expected.

Finally, the ultimate challenge is in validating the system as a whole. This includes verifying the system whilst running the application software. Unfortunately, conventional simulation based validation is not fast enough to execute the millions of vectors required to run even the smallest fragment of application code. There are three approaches to address this issue. First, using specialised hardware for performing validation. Second, increasing the level of abstraction which increases the simulation speed. Third, using formal verification which eliminate the use of test vectors. Although the first approach is the preferred option, the cost implication forces many design houses to opt for using the second or third approaches as much as possible.

3.6 Summary

SoC technology offers new challenges and opportunities. On the one hand, it may be the answer for easing the increased pressure on time-to-market, and reducing the design and manufacturing cost. On the other hand, it is recognised that SoC design methodologies are still immature. The design methodology is still evolving to cope with evolving issues. There is a need to define common standards for the SoC design environment so that it can foster design development. The issue of exchange and inter-operability of cores will require development, so that building an SoC design can become as easy as building a board design. The component's evaluation and selection will be at least as significant as the design of custom logic. The complexity of an SoC design will lead to a major bottleneck in system level functional simulation. Finding a way to facilitate system-level validation will be a major challenge.

Chapter 4

Integration and Verification Platforms For SoC Designs

In this chapter, the discussion is focussed on the design methodology used in the creation and validation of an SoC design. In Section 4.1, the methodology used in SoC design is explained. The standard architecture for creating an SoC application is presented in Section 4.2. In Section 4.3, a formal verification methodology for SoC design is introduced. The architecture of a formal verification system is briefly described in Section 4.4. The chapter is concluded with a summary.

4.1 Integration Platform

Platform-based design emerges as the methodology to develop SoC designs. In this methodology, the designers integrate block designs or modules which originated from various sources into a standardised platform. A standardised platform will allow for rapid system development and is extensible for the development of derivative design applications. This standardised platform is also called the *integration platform*. By definition, an integration platform is an environment to develop an SoC design by mixing and matching reusable virtual components. An integration platform is defined by targeting a specific application domain [54].

An integration platform consists of:

- The SoC integration architectural specification. It describes the basic architecture of the hardware such as the bus structure, the test architecture, the I/O configurations, the required VC blocks, and the constraints (performance, power, and area). The description may also include the basic software architecture such as the *real time operating system* (RTOS), task scheduler, inter-task communication, software/hardware communication, and the middleware and application software structure.

- The portfolio of virtual components which meet the constraints defined in the integration platform. These VCs form a collection of component libraries for a specific targeted application domain and are used in the mix-and-match integration process.
- The documented design methodologies for block authoring and chip integration. The methodologies include the support for the design, verification, planning and integration of new logic or VCs.
- A design verification methodology and environment. The methodology addresses the problem of hardware/software co-verification and specific verification requirements for the application domain.
- A prototype characterisation of the behaviour of the integrated architecture.

The practice of using a platform to create a design will lead to a fundamental shift in the design methodology. This may include the following:

- The reusable platform to increase productivity and re-use will restrict possible derivative applications. It is targeted only for derivation within the scope of an application domain. This restriction will allow for a higher degree of assurance that the system will work when the components and the architecture are integrated together.
- There is a shift in design creativity towards creating the platform. The integration process itself is similar to creating a board design with a set of prefabricated IC components. The restricted derivative application makes the design integration process low-risk, predictable and fast.
- The integration platform is built around a processor or a DSP core and a specific bus architecture. These two components are the core modules, whereas the other components such as peripherals are added and evolved to meet the requirements of the application.

4.2 Integration Platform Architecture

One key problem in SoC is the infrastructure for communication between cores. The *Virtual Socket Interface Alliance* (VSIA) has developed specifications for an on-chip bus interface to solve this problem. However, various vendors have created their own proprietary communication protocols associated with their own cores. This has made standardisation a difficult task. Examples include the *Open Core Protocol* (OCP) from Sonics Inc [73] and the *Advanced Micro-controller Bus Architecture* (AMBA) from ARM [7]. Sonics's OCP bus protocol is based on a single bus architecture while ARM's AMBA bus protocol is based on multi-bus architectures. Of the two specifications, the AMBA specification is used more in the building of embedded SoC systems. The rest of this section describes the multi-bus integration platform architecture.

For some companies, such as Texas Instruments (TI) and VLSI Technology, SoC is not a new activity. They have both developed their own integration platforms for wireless systems. TI developed the *TI standard-independent platform* for single-chip digital baseband wireless system. The platform combines the use of several modules such as TI-DSP, the ARM7 processor, RAM/ROM, analog blocks, and logic blocks [79]. VLSI Technology has also developed a communication standard platform, which combines OAK-DSP, the ARM7 processor, RAM/ROM, analog blocks, and logic blocks [83]. Both companies use similar basic components and they incorporate third party IP blocks as part of their design.

Similar to TI and VLSI, three major providers of EDA tools Cadence [17], Synopsys [41], and Mentor Graphics [49], have proposed a similar idea of system integration platform. A typical integration platform architecture is presented in Figure 4.1 [64]. A general-purpose processor core is the basic component in the integration platform. All the IP blocks are glued together through busses that communicate with the processor. Within this environment, two kinds of bus 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 the PLB communications amongst the processor, memory and OPB bridge. The OPB arbiter controls the OPB communications amongst the IP blocks. The IP blocks can be user-defined logic blocks or third party IP blocks.

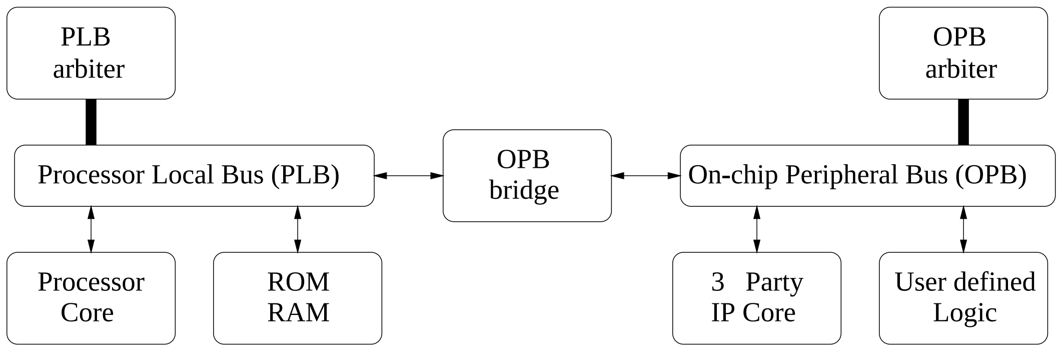

Fig. 4.1: Integration Platform for System on Chip

4.3 Verification Platform

In parallel with the system integration platform described above, this dissertation encourages the construction of a *formal verification platform*. 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 [77, 78]. In this, each of the building blocks is represented as a formal specification model (Figure 4.2). There is a model for the processor core, for the bus and its protocol, and for all IP blocks available. The different models are integrated as a single system description in the verification platform.

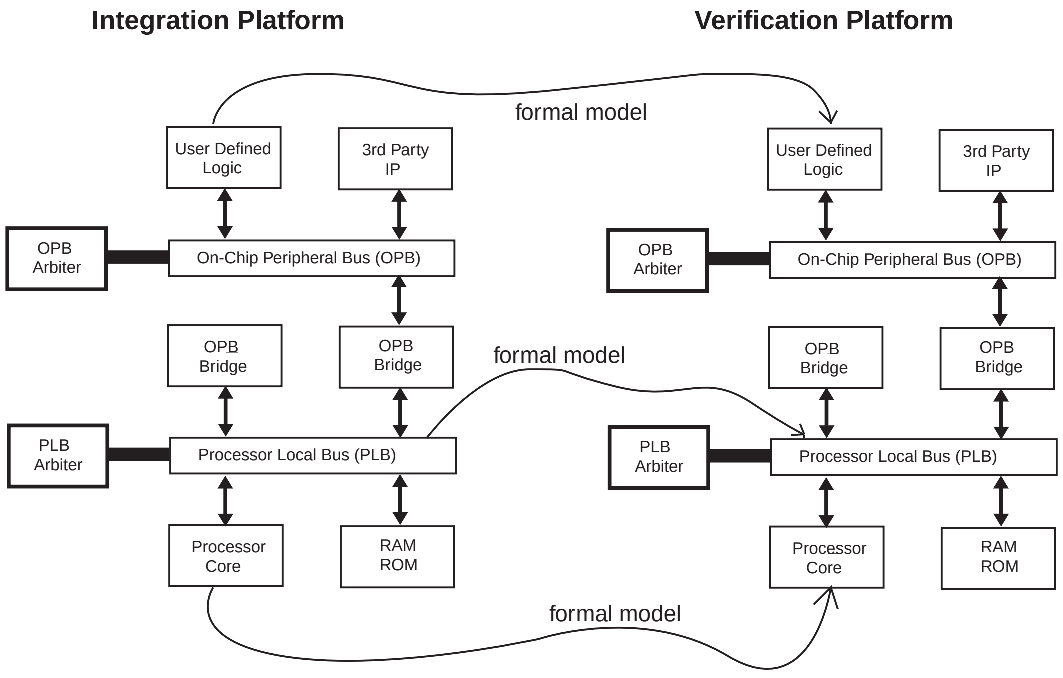

Fig. 4.2: Verification Platform for System on Chip

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 [84]. A bus protocol formal model is normally verified using a property checker [72]. The verification platform needs to accommodate all these various verification techniques.

The formal verification platform is created through the following steps:

- The analysis of component behaviours. The results are used as a guideline to decide which formalism is best suited to model the component. For example: protocols are best suited to be modelled in a model checking environment, whereas processors are best modelled in a symbolic simulation environment.

- The creation of formal models for each component. Components are modelled in a relational modelling style. In this modelling style, the behaviour of a system is built by wiring together components by conjoining the predicates. Higher Order Logic is used as the glue logic to combine and connect formal components. This approach is similar to the way a system is created in the integration platform. If a formal component is functionally modelled, an interface to transform the functional model to a relational one is needed.
- The development of generic behavioural properties for each formal model. Obtaining the generic form for the properties enables them to be used without the need to redo the verification every time it is used to create a new design. One approach to obtain the generic properties is by defining the properties as input/output relations.
- The system level verification of the formal verification platform. The verification uses the generic properties of the components and combines them to obtain the system level properties. The verification process is conducted using the most appropriate tool available in the formal verification environment.

4.4 Formal Tools Architecture

There are two approaches to define 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: equivalence checking, model checking, and theorem proving. HOL [34, 80], PVS [74], ACL2 [14, 47, 48], and Forte [2, 3, 45] are examples of this kind of verification environment. Another approach is to use a mixture of available tools, PROSPER [27] being one notable example of such an environment. 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 problem 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.

This work uses the mixed tools environment. We construct *a verification environment* which has the capabilities of various formal verification

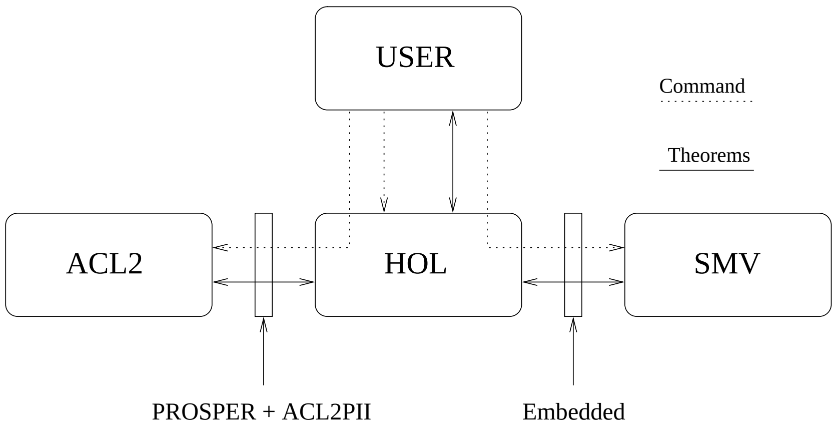

Fig. 4.3: Heterogenous Formal Tools System

technologies, such as a *symbolic simulator*, a *model checker*, and a *theorem prover*. The verification environment combines the HOL98 theorem prover [80], the ACL2 theorem prover and symbolic simulator, and the SMV model checker [56, 57]. The architecture of the formal tools system is presented in figure 4.3.

HOL98 is the centre of the tools environment. ACL2 and SMV are connected to HOL98 through a layer of interfaces. The communication between ACL2 and HOL is managed by PROSPER and ACL2PII. The SMV model checker is embedded in HOL as one of the decision procedures for HOL's tactic language. 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.

The system provides an environment such that every user can interact with the tools in their original environment. Furthermore, the platform provides an integrated control environment controlling ACL2 and SMV through the HOL98 environment. A detailed description of each formal tool used in the verification environment and the interface between the tools are explained in the next chapter.

4.5 Summary

The formal verification platform concept is an adaptation of the integration platform concept. In the formal verification platform, a formal system is created by integrating various formal models and is then formally validated. Central to this concept is the formal verification environment where various formal tools are integrated to create a complete set of formal techniques. The verification environment combines the capabilities of the HOL98 theorem prover, the ACL2 theorem prover, and the SMV model checker.

Chapter 5

The Formal Verification Environment

In this chapter, descriptions will be given of the *Formal Verification Environment* (FVE) used in the rest of the dissertation. The descriptions are split into two main categories: the description of individual formal tools, and the integration of the formal tools.

In Section 5.1, a brief introduction to the formal hardware verification technologies is presented. The following three sections give a brief introduction to the formal verification tools used in FVE and the approach used to create formal models. The ACL2 theorem proving system is described in Section 5.2. The HOL theorem proving system is described in Section 5.3. The SMV model checker is described in Section 5.4. The integration of the ACL2 theorem prover and the HOL theorem prover is presented in Section 5.5. The integration of the SMV model checker and the HOL theorem prover is presented in Section 5.6. Finally, the summary of this chapter is presented in Section 5.7

5.1 Introduction to Formal Hardware Verification

Design validation involves taking steps to guarantee that a design will perform according to its specification. The process of validating a design can be grouped into three stages: pre-implementation, implementation, and post implementation [51]. In the pre-implementation stage, checks are made as to whether the design is as intended. In the second stage, checks are made as to whether the design has actually been implemented. In the final stage, the design is manufactured and the product is tested. Formal hardware verification is concentrated in the first two stages. Validation of the pre-manufacturing stage is based on a brute-force simulation technique, either by running a number of independent simulations distributed over a set of machines, or using special purpose simulation hardware. However, despite these tremendous efforts, serious design errors often remain undetected. Formal hardware verification attempts to overcome the weakness of non-exhaustive simulation.

Formal methods are the application of mathematical methodologies to the specification and validation of systems. Formal methods and their applications to perform correctness proofs are well-known and have been in existence for many years. Originally they were intended to verify software. Unfortunately, it is not used as a standard technique in the verification of software components. On the other hand, hardware verification has been explored for just two decades, but commercial tools are already available and hardware verification is now becoming a standard for design flow.

There are several reasons why formal methods have been quickly adopted in circuit design. First, the quality standard for hardware is much more rigid compared to software. Errors in hardware are very expensive to correct. Moreover, there is no customer tolerance for faulty designs as hardware cannot be easily replaced or corrected like software where patches for correcting the system are easily provided, or even an upgrade to a new version of software. Second, hardware design as an engineering discipline enforces a structured and well-defined design cycle. This comprises a unified refinement process using standard abstraction levels and a high degree of reuse. The reuse of already designed components either justifies a costly verification or allows for a cheaper verification, since previously employed verification strategies can be used again. Third, hardware designs are considered to be simpler compared to software. The nature of hardware forces the use of finite domain data-types (bits and bit vectors) which makes it possible to use a high degree of automation. In many cases the structure of the design can be directly used to automatically steer a hierarchical verification approach.

Formal hardware verification technologies can be grouped into three categories: equivalence checking, model checking, and theorem proving [25, 71]. Equivalence checking is a technique used to verify the equivalence of a reference and a revised design. The tools perform an exhaustive check on the two designs to ensure that the designs behave identically under all possible conditions. Model checking is a verification technique that is based on building a finite model of a system and checking that a desired property holds in the model. It is an automated technique used for verifying finite state concurrent systems by performing an exhaustive state space search. Theorem proving is a technique where both a system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas.

A formal verification environment which offers all of these formal technologies is built by integrating three formal verification tools: the ACL2 theorem prover, the HOL theorem prover, and the SMV model checker. The combination of these three formal tools enables the user to perform a combination of verification techniques. Using this formal verification environment, formal analysis can be performed on SoC circuits.

5.2 ACL2 Theorem Prover

ACL2 stands for *A Computational Logic for Applicative Common Lisp.* It is a theorem proving system and a programming environment. The logic of ACL2 is based on quantifier-free *First Order Logic.* The ACL2 language is built as a variant of the Common Lisp language [86]. It is described as a mathematical logic with axioms and rules of inference. Functions and formal models specified in ACL2 are used as specifications and as simulation engines. In a programming environment, functions and formal models can be executed or simulated. As specifications, they can be reasoned about to prove properties or theorems about them.

Functions in ACL2 are defined using the *defun* expression. With concrete arguments, these functions can be evaluated. At the same time, they are also represented as logical objects about which one can reason. For example, an ACL2 addition function (*plus*) is described in Definition 1.

Definition 1 (plus)
```
(defun plus (a b) (+ a b))
```

The defun expression defines plus as a function that takes two arguments and returns the result of adding the two inputs. When the function is evaluated with concrete values, such as (plus 4 5), it returns 9 as the result. Some primitive functions of ACL2 are described in Table 5.1

Lisp is syntactically untyped. When LISP function evaluates arguments outside the intended domain, the result depends on the implementation. ACL2 provides a mechanism to specify and restrict the arguments to the intended domain. When the evaluation goes outside the domain, an error message is signalled. The intended domain of a function is specified by its *guard.* For example, the guard of the plus function can be specified for natural numbers. The new plus function (*gplus*) is defined in Definition 2. The function accepts only natural numbers as arguments for evaluation.

Definition 2 (gplus)
```
(defun gplus (a b)
  (declare (xargs :guards (and (natnp a)(natnp b))))
  (+ a b))
```

ACL2 Function and Constants	Description
T	True value
nil	False value or empty list
(and p1 p2)	Logical conjunction operator
(or p1 p2)	Logical disjunction operator
(implies p q)	Logical implication
(not p)	Logical negation
(iff p q)	Logical equivalence
(acl2-numberp x)	Recogniser for any type of ACL2 number
(integerp x)	Recogniser for integers
(rationalp x)	Recogniser for rationals
(zerop x)	x=0
(zp x)	x=0 or x is not a positive integer
(< x y)	Less than relation
(<= x y)	Less than or equal relation
(> x y)	Greater than relation
(>= x y)	Greater than or equal relation
(+ x y)	Addition
(- x y)	Subtraction
(/ x y)	Division
(1- x)	Decrement by 1
(1+ x)	Increment by 1
(characterp x)	Recogniser for character
(consp x)	Recogniser for ordered pairs
(cons x y)	Construct an ordered pair
(car pair)	First component of a pair
(cdr pair)	Second component of a pair
(atom x)	Recogniser for non-pairs
(list x_1 ... x_n)	Linear list of objects
(nth n lst)	n^{th} element of a list.
(len lst)	Length of a list
(true-listp x)	Recogniser for linear lists

Tab. 5.1: Description of basic ACL2 functions and constants

Lemmas and Theorems in ACL2 are defined with the *defthm* expression. For example, the result of evaluating the *gplus* function will always produce a natural number. Given two natural number variables *a* and *b*, the symbolic evaluation of *gplus* on the variables results in an addition operation of the variables. Theorem 1 (*gplus-gives-natnp*) describes the proof of the *gplus* function.

Theorem 1 (gplus-gives-natnp)
```
(defthm gplus-gives-natnp
 (implies (and (natnp a)
               (natnp b))
          (natnp (gplus a b))))
```

The ACL2 system allows users to perform *symbolic simulation*. In symbolic simulation, the user can execute a design on certain kinds of indeterminate data. This allows the coverage of more cases in a single execution. With symbolic simulation we can evaluate the *gplus* function symbolically. The symbolic evaluation of *gplus* function is described in Theorem 2.

Theorem 2 (symbolic-simulation-of-gplus)
```
(defthm symbolic-simulation-of-gplus
 (implies (and (natnp a)
               (natnp b))
          (equal (gplus a b)
                 (+ a b))))
```

Undoubtedly, the *symbolic-simulation-of-gplus* theorem is a trivial one. However, the simplicity also holds when a complex function is executed, such as a processor model, with program instructions. We use this feature to symbolically simulate a fragment of code which is executed using a processor model.

In ACL2 logic, there is no distinction between lemmas and theorems. The ACL2 theorem prover exploits mathematical induction and term rewriting with heuristics to prove theorems automatically. In the case of a complex theorem, the user may have to provide the outline of the proof by providing intermediate theorems. The collection of these intermediate and final theorems is called a *book*.

A digital circuit is modelled in ACL2 using a functional style of specification. In this style, the output of every component in the design is specified as a function of inputs to the component. An example of the functional style of specification is as previously described in Definition 1.

5.3 HOL Theorem Prover

HOL is a *Higher Order Logic* theorem prover. The HOL logic is based on higher order predicate calculus which allows variables to range over functions and predicates. The HOL logic is built on an ML style typed system [62]. A term in HOL logic is represented by an abstract data-type called *term*. HOL has four kinds of primitive term: *variables*, *constants*, *function applications*, and λ-*abstraction*. A variable is a term which has a name and a type. The definition of a constant is similar to a variable, except it has a fixed type and cannot be bound by quantifiers. A function application is a term with the format of (f t), where f and t are terms. It has types of the form $\sigma_1 \rightarrow \sigma_2$, where σ_1 and σ_2 are the types of the domain and range of the functions. The λ-abstraction is a term with the format (λx. t), where x is a variable and t is a term. In the HOL system, the symbol '\' is used to represent λ. For example, (\x. x+1) is a term that denotes the function $n \rightarrow n{+}1$.

The HOL system uses these primitive terms in defining its basic terms. The truth and falsity terms are Boolean typed constants and defined as T and F. The function applications such as the negation ($\neg$) is a term with the type (bool $\rightarrow$ bool). Some functions, such as ==>, \/, /\, etc., are infixes. They have special syntactic status which allows them to be written as infix expressions, e.g. (t_1 ==> t_2). The basic HOL terms are presented in Table 5.2

Kind of term	HOL notation	Description
Truth	T	True value
Falsity	F	False value
Negation	$\neg$ t	Not t
Disjunction	t_1 \/ t_2	t_1 or t_2
Conjunction	t_1 /\ t_2	t_1 and t_2
Implication	t_1 ==> t_2	t_1 implies t_2
Equality	$t_1 = t_2$	t_1 equals t_2
$\forall$ quantification	!x.t	forall t
$\exists$ quantification	?x.t	exists t
ε term	@x.t	an x such that t
Conditional	(t => t_1 | t_2)	if t then t_1 else t_2

Tab. 5.2: Description of basic HOL terms

A function in HOL is defined using a *Define* expression. *Define* declares a new constant (*plus*) and installs a definitional axiom (*plus_def*) in the current theory. The function *plus* is described in Definition 3.

Definition 3 (plus_hol)

```
Define ‘(plus a b = a+b)’;
Term constants:
    plus :num -> num -> num
Definitions:
    plus_def |- !a b. plus a b = a + b
```

One way to prove a theorem is by proving the goal with a tactic. A goal of a theorem is the target of the proof which defines the validity of the theorem. A tactic is an ML function that when applied to a goal reduces the goal into sub-goals. If the proof succeeds, it returns a theorem which can be stored in the current theory.

In HOL, a digital hardware circuit can be specified in two modelling style: the functional and the relational styles [58]. The functional style specification is as specified in Definition 3. In the relational style specification, digital hardware circuits are specified in higher order logic by defining their predicates. The predicates state the combinations of values that can appear on their external input and output ports. The behaviour of devices are built by joining smaller devices together by connecting them at all identically-labelled external wires. The composition is done with logical conjunction and using existential quantification to hide internal wiring.

As an example, consider an inverter circuit. The circuit diagram is shown in Figure 5.1(a). The inverter has one input port (i) and one output port (o). The variables i and o have the logical type *bool*. The term Inv(i,o) describes a relationship between the variables. The output o becomes true when i is false, and vice versa. The relational model of Inverter is specified in Definition 4.

Definition 4 (Inverter)
$\vdash$Inv(i,o) = (o = $\neg$ i)

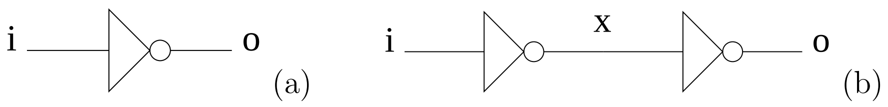

Fig. 5.1: (a) An inverter (b) Wiring two inverters

In Figure 4(b) two inverters are joined together by an internal wire x. The combined models are specified by applying logical conjunction to the components and hiding the internal wire x by existentially quantifying the wire. The circuit is described in Definition 5.

Definition 5 (Two Inverters)
$\exists$x. Inv(i,x) $\wedge$ Inv(x,o)

5.4 SMV Model Checker

SMV is a model checking tool, a tool for automatically checking the validity of temporal logic formulas in finite-state circuit models. Digital hardware circuits are represented as finite-state models and in a symbolic form that uses *Binary Decision Diagrams* (BDDs) to give a compact representation. The finite model is defined in a state transition graph called a *Kripke structure*. It consists of a set of states, a set of transitions between states and a function that labels each state with a set of properties that are true in the corresponding state. The computation of the system is modelled as paths. A specification for SMV is a collection of properties. A property states the relationship between the values or timing of the signals. Properties are specified in a notation called *Temporal Logic* [23]. The Cadence SMV model checker supports *Linear Temporal Logic* (LTL) specifications.

LTL is based on the idea that time is linear. It considers only one future or execution which has actually taken place. An LTL formula is an assertion about one particular sequence of states. The formula is similar to an ordinary boolean logic formula, except the truth value of a formula is a function of time ($num \rightarrow bool$). New operators to specify relationships in time are added to the existing boolean logic operators (and, or, not, implies).

The future (**F**) operator is used to express that a condition is true at some time in the future. The formula ($\mathbf{F}\varphi$) is true at time = 0 (t_0) if φ is true at some time later ($t \geq t_0$). ($\mathbf{F}\varphi$) is usually read as *eventually* φ.

The globally (**G**) operator is used to state that a condition is true at all times in the future. The formula ($\mathbf{G}\varphi$) is true at t_0 if φ is always true at ($t \geq t_0$). ($\mathbf{G}\varphi$) is usually read as *always* φ.

The next (**X**) operator is used to express that *next time* a condition is true. The formula $\mathbf{X}\varphi$ is true at t_0 if φ is true at *($t = t_0$+1)*.

The until (**U**) operator is used to express that a conditional condition is true at some time in the future. The formula ($\varphi\mathbf{U}\psi$) is true at t_0 if eventually ψ is true at t_x and φ is true at ($t_0 \leq t < t_x$). The visualisation of these four LTL operators are presented in Figure 5.2

SMV uses the SMVL (SMV language) to specify a model. The SMVL can be divided into three parts: type declarations, signal assignments, and assertions. Type declarations define the type of every signal used in the system. The signal type can be in the simple form of boolean, enumerated type (for example {ready,wait}), and subrange (for example 0 .. 7) or an abstract data-type such as *undefined*. The *undefined* data-type tells SMV that the type is a symmetric

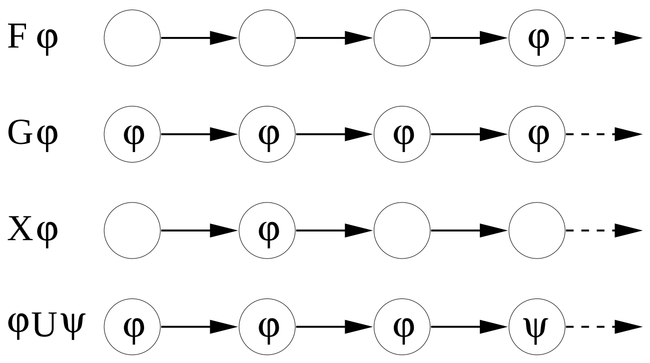

Fig. 5.2: The visualisation of LTL operators

type, but does not state what the values of the type are. The signal assignments define the relationship between input and output signals. An assertion is a condition that must hold true in every possible execution of the program.

SMV also accepts circuits described in *Verilog Hardware Description Language* (Verilog HDL). When SMV reads the Verilog code, it transforms the code into SMVL using a transformation tool called *vl2smv*. This feature enables the users to directly write the models in an HDL style rather than specify them in SMVL.

The SMV model checker automatically verifies the hardware circuit's properties by interacting with the system's finite state machine. The verification searches all possible states that the system can reach. In some cases, the SMV verifier produces a counter-example which is a behaviour trace of the finite state system which violates a specified property. This counter-example helps users to debug the circuit.

5.5 ACL2-HOL Integration

The ACL2 and HOL theorem provers use different languages and different logics. The ACL2 uses untyped *s-expressions* (s-exp) to represent first order logics, whereas the HOL uses typed terms to represent higher order logic statements. To combine these two theorem provers, we need an interface which can bridge these two worlds. One possible solution is by integrating ACL2 into the PROSPER [27] framework.

PROSPER is an environment for building a custom verification engine which can be operated by another application through an *Application Programming Interface* (API). The proof engine is based upon the functionality of a theorem prover with additional capabilities provided by *plug-ins* from existing off the shelf tools. The PROSPER toolkit comes with two main components: the *PROSPER*

Integration Interface (PII) and a *Core Proof Engine* (CPE) . The PII is a set of libraries which enables communication between components. The CPE is the main proof engine where other proof engines (such as theorem provers, model checkers, and decision procedures) can be integrated as plug-ins. The CPE command language is used as the glue language for managing plug-in components and orchestrating the proof engines. The PROSPER toolkit is implemented around the HOL98 theorem prover. Through the PROSPER framework, HOL can interact with external systems such as ACL2.

The communication link between ACL2 and PROSPER is provided by the *ACL2 PROSPER Integration Interface* (ACL2PII) [76]. ACL2PII is a dynamic link for translating theorems between two *live sessions* of HOL and ACL2. Both ACL2 and HOL communicates to each other via ACL2PII and PROSPER. The interface allows a user to run ACL2 from within HOL. It also enables ACL2 to pass or export its knowledge of a particular ACL2 book into HOL. Through this link, ACL2 is being used as an axiom-server which produces axioms about constant that are otherwise uninterpreted in HOL. The logical consistency of the resulting HOL theory is assured by the proofs having been done in ACL2 and by an understanding about the correspondence between any background theories used in each of the tools.

The ACL2PII is based on an *ad-hoc* scheme for translating ACL2 s-expressions to HOL terms. A set of default translations is available such that appropriate s-expressions can be translated into Booleans, Natural numbers, integers, simple arithmetic expressions, strings, characters, lists and tuples. These are the basic constructors in the interpretation of ACL2 s-expressions in the environment of the standard HOL library. For example, Boolean in ACL2 is the same as Boolean in HOL and the constant $\wedge$ in HOL is the same as the AND operator in ACL2.

ACL2PII provides a method for specifying new translations. It is by defining a translation pattern based on a triple of a string which represents an ACL2 s-exp, a HOL term quotation, and a list of HOL term quotations representing side-conditions on the translation. The ACL2 string is used to do pattern matching of the ACL2 pattern with the actual ACL2 s-expressions. Consider these two translations:
(1) (“(− X Y)”, ‘X – (Y:int)’, [ ])
(2) (“(− X Y)”, ‘X – (Y:num)’, [‘Y <= X’])
The two subtraction translations are similar, except that the first one is intended for integer subtraction while the second one is for natural number subtraction. The infix subtraction of the HOL term ‘X – Y’ is a constant and X and Y are the variable names. These variables are used as variables for matching in the ACL2 s-exp (− X Y). The s-exps of X and Y are recursively translated into HOL terms and substituted for X and Y in the HOL term pattern. In the first translation,

the substitution pattern of Y is an integer. The variable X is automatically defined as an integer. The empty list indicates that there is no side condition for integer subtraction. In the second translation, variables X and Y are defined as natural numbers. When the value of X is less than the value of Y, the ACL2 evaluation will be a negative number while in HOL the result will be zero. To avoid inconsistency of the translation process, a side condition of 'Y <= X' is added. When an ACL2 theorem is imported into HOL, the side-condition is added as part of the hypotheses on the theorem's conclusion.

ACL2PII provides the *mkbasefun* function to declare a HOL constant which corresponds to an ACL2 function. The function creates an automatic translation and a type guessing function for the new constant. The *mkbasefun* function takes four arguments. The first and third arguments represent the ACL2 function name and the HOL constant. The second argument is the number of inputs for the functions. The last argument is the type definition for the constant. For example, the plus function (Definition 3) is translated into HOL with (mkbasefun "plus" 2 "acl2plus" (Type ':num→num→num')). The function creates a HOL constant *acl2plus* which has the type *num→num→num*. If it is not needed, the ACL2 definition of *plus* does not need to be imported into HOL, although it is possible to import ACL2 definitions into HOL. The automatic mechanism is provided by ACL2PII's *mkfun* function. The *mkfun* function only requires three arguments which is similar to *mkbasefun*, except it does not need to know the number of inputs. Executing (mkfun "gplus" "ACL2GPLUS" (Type ':num→num→num')) creates a HOL constant *ACL2GPLUS* and translates the ACL2 function definition *gplus* to create a HOL equality *definition* theorem. The ACL2PLUS function definition is as follows:

Definition 6 (ACL2GLPUS_hol)
∀ A B. ACL2GLPUS A B = A + B

ACL2 theorems are imported into HOL using the *getthm* function. This function recovers the theorem from ACL2 and translates it into an HOL term. The term is then axiomatised as a theorem using HOL's oracle mechanism. Executing (getthm [] "gplus-gives-natnp") results in a HOL theorem described in Theorem 3.

Theorem 3 (gplus-gives-natnp_hol)
((natnp a ∧ natnp b) → natnp (gplus a b))

It is also possible to execute an ACL2 function with a concrete instance from HOL. The mechanism is provided by the *getexec* and *command* functions. The *getexec* function takes an ML term and executes it in ACL2. The ACL2 execution result is parsed and translated into HOL as the right hand property of the equivalence operator. Then it is asserted as a theorem using HOL's oracle mechanism. The *command* function performs a similar task, except it takes a LISP s-expression. Executing (command "plus 3 4") results in a theorem ($\vdash$ (plus 3 4) = 7).

A detailed description of the PROSPER toolkit and ACL2PII is presented in [27] and [76] respectively.

5.6 SMV-HOL Integration

The HOL98 distribution includes an early version of McMillan's SMV symbolic model checker as part of the temporal logic library. LTL is embedded in HOL using the shallow embedding technique. The temporal operators are defined as functions that are applied to arguments of type (num → bool). For example: the NEXT operator is defined as a new definition in HOL. The function accepts one argument P which has the type (num → bool). The HOL description of NEXT is as follows:
val NEXT = new_definition("NEXT", - -'NEXT P = \t. P(SUC t):bool'- -);
The meaning of the definition is as follows: (NEXT P) is true at time t when P is true at time (SUC t) or (t + 1).

In the library, the model checker is embedded in HOL as one of the decision procedures for HOL's tactic. Using this library, temporal properties are specified in LTL notations. Note that the model checker which comes with the distribution only supports CTL (Computation Tree Logic) style temporal logic. The model checker can not directly deal with properties specified in LTL. The LTL formulae need to be transformed into a format which can be understood by the model checker.

It is not possible to directly translate LTL formulae to CTL formulae. But it is possible to convert LTL formulae into an intermediate format, namely a finite state ω-automata [69, 70]. The translation of the formula to this format can be viewed as transforming the problem into a fix-point one. While CTL is a subset of the alternative free μ-calculus which enables the reduction of model checking problems into fix-point computations. The LTL formula which has been converted into ω-automata can be validated in two ways, either by proving the properties using traditional HOL tactics or by using an external model checker. When a model checker is used and the formula can be verified then the result from the model checker is represented as a theorem using HOL's oracle mechanism. If the model checker reports an error, then a counter example is provided.

The latest version of the SMV model checker was developed at CADENCE. This re-implementation of SMV uses LTL instead of CTL. Although for backward compatibility it supports CTL style, the developers suggest the use of LTL style to achieve maximum performance. The temporal library is extended in this instance so that it is possible to use the CADENCE SMV model checker in LTL notations. A subset of SMVL is embedded in HOL using the deep embedding technique. In deep embedding, the semantics of the language is constructed and an interpretation of the language is provided. It also makes the system more modular. Previously, when a model checker was used, the formal model had to be specified in HOL. Now, the SMV model can be verified on its own and the proved properties are automatically imported as HOL theorems.

5.7 Summary

In this chapter, a gentle introduction to formal verification technologies commonly used in formal hardware verification was given. In general, the technologies are grouped into three categories: equivalence checking, model checking, and theorem proving. Each technology is uniquely used in different aspects of formal hardware verification. Verifying an SoC design as a whole combines all verification problems which are commonly dealt with separately.

An integrated heterogenous formal verification environment was created which represents those three technologies. The system is based on the ACL2 theorem prover, the HOL theorem prover, and the SMV model checker. Physical interfaces that combine the tools are needed. The HOL theorem prover is at the center of the proposed formal system where users control the verification process. The ACL2 theorem prover utilizes the PROSPER framework as its interface to communicate with the user. The SMV model checker is embedded as one of HOL's decision procedures.

The formal models can be specified in two modelling styles: in functional style or in relational style. ACL2 and HOL allow formal models to be specified in functional style. The relational formal model can only be specified in HOL. The integration of formal models is done in HOL using the relational modelling style.

Chapter 6

The Simple Integration Platform (SIP)

Chapter 5 discussed the heterogenous formal verification environment. The environment is used to specify components of SoC design in the most suitable formalism. In this chapter, the formalisation of an SoC design using the formal verification environment is demonstrated. As an example, a system called SIP is defined. The formalisation of SIP will provide a concrete example of the formal verification platform concept introduced in chapter four. It will also provide a concrete illustration of how links between heterogenous collection of tools and logic systems in the platform actually work in practice. The whole process will lay the foundation of how to model and verify SoC designs.

The introduction of SIP architecture is presented in Section 6.1. In Section 6.2, the specification of the processor module is described. Section 6.3 describes the Interrupt module. The specification of the memory module is described in Section 6.4. Section 6.4.3 describes an example of a software component used in the verification. In Section 6.5, the bus-controller and bus-multiplexer will be defined. Finally, the integration of all modules will be presented in Section 6.6. A summary will conclude this chapter.

6.1 Introduction

Simple Integration Platform (SIP) is an integration platform based on the architecture proposed by Richard Black in [13]. In this case study, every module in the platform is represented by its simplified version. The use of the full specification modules would only increase the verification complexity without adding a new concept. Furthermore, the simplified version will provide a better clarity of Sthe proposed verification methodology.

The bus protocol, which controls all communications between master and slave modules, is specifically designed for the architecture and without a standardised protocol methodology [5]. An extended version of SIP will be explored in the case study presented in Section 9.3. In contrast with SIP, the later case study will use industrial standard components as its building blocks. The architectural similarity between the two platforms will allow the verification methodology of SIP to be used as the guideline in the verification of the extended one.

The SIP system contains four modules: two master modules, one slave modules, and one SIMBC (*System Integration Module and Bus Control*) module. The processor and the interrupt are the master modules. The memory is the slave module of the system. The SIP architecture only has one data-bus. All data transfer between master and slave has to go through this bus. The block diagram of the SIP architecture is described in Figure 6.1.

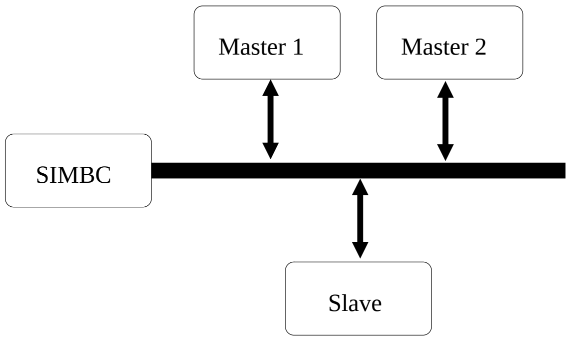

Fig. 6.1: Simple Integration Platform Block Diagram

One of the masters is set as the default master. It has the feature of always requesting the bus and always being granted when no other masters send their request signals. Furthermore, it has the capability to lock the bus. In this condition, the bus-controller will ignore any other bus request until the lock signal is withdrawn. The second master has a higher priority than the default master. In most cases, it will immediately get the bus. The only exception is when the bus-controller is serving a bus-lock request. There are interdependency requirements between these two modules in granting their bus access request. This case study will show how the whole system works together in satisfying and reducing the requirements.

The choice of how to formalise each component depends on what properties will be proven and what is the best formal technology to verify them. In this case study, two properties of SIP system will be verified: the liveness of every master request and the correctness of a software component executed in the SIP system. First, the liveness verification concentrates on the verification of the bus protocol. The properties are normally verified using a model checker. The SIMBC will be modelled in the model checking environment. Second, the correctness of a software component is normally conducted using a simulation technique. Based on this, the software component and its execution engine are modelled in the environment where symbolic simulation can be performed. More detailed descriptions of the properties being verified are presented in the next chapter.

6.2 Processor Model

6.2.1 The Architecture

The processor is the default master module. It features a three-stage pipeline architecture: fetch, decode, and execute. In this architecture, before an instruction is evaluated in the execute stage, it has to go through the fetch and decode stages. The independence of each pipeline stage allows the processor to operate on three different instructions in the same clock cycle. Consider three instructions ($Inst_1$, $Inst_2$, and $Inst_3$) which will be executed by the processor. At time t, the processor fetches an instruction (Inst_1) from an external memory module and stored it in the third pipeline register (p3). At time (t + 1), Inst_1 is moved to the second pipeline (p2) and is decoded. At the same time, a new instruction (Inst_2) is fetched and stored in p3. At time (t + 2), Inst_1 is moved to the top pipeline register (p1) to be executed. Meanwhile, Inst_2 is moved from p3 to p2 and starts the decoding stage. At the same time, a new instruction (Inst_3) is being fetched and stored in p3.

The processor has eight internal registers. It is based on a nine instruction set architecture. These instructions are capable of performing program branching, data manipulation, and data transfer operations.

Interfaces

The processor has three bus lines (*PDataIn, PDataOut, PAddress*). It has five control signals, two input signals (*PnReset, PnWait*), and three output signals (*PLock, PnRW, PnMreq*). The description of the interfaces are presented in Table 6.1.

Internal Registers and Flags

The processor has eight internal registers (r_0 - r_7). Registers r_0 to r_5 are the *General Purpose Registers* (GPR). Registers r_6 and r_7 are the *Special Function Registers* (SFR). Register r_7 stores the value of the *Program Counter* (PC). When the processor is executing data transfer operations, it changes the address to a new value. The processor has to save the current address so that it can resume the process before the address changes. The old address is saved in the temporary PC register r_6. After the processor completes the data transfer process, it restores the address to the one saved in r_6.

Name	Description
PnReset	is the reset signal. A LOW level input signal forces the processor to go to the default reset state.
PnWait	is the wait signal. A LOW level input signal stalls the processor.
PLock	is the lock signal. A HIGH level output signal indicates that the processor is executing a locked memory access.
PnRW	is the read or write signal. The processor is in the read cycle when the signal is LOW. Otherwise, it is in the write cycle.
PnMreq	is the memory request signal. When the signal is LOW, it indicates that the processor requires bus access.
PDataIn	is the input data line. This input line allows the data to go into the processor.
PAddress	is the address line. This output line provides the address that the processor is accessing.
PDataOut	is the output data line. This output line provides the data to be transfered out of the processor.

Tab. 6.1: Processor Input/Output Interfaces

The processor has one flag: the zero flag *ZF*. *ZF* is updated when the processor is executing data processing instructions. When the result of data manipulation is zero, *ZF* is set to HIGH. Otherwise, it is set to LOW.

Instruction Sets

The processor implements nine instructions. Based on how it operates, these instructions can be grouped into four categories: no operation instruction (NOP), conditional branch instruction (*JZ*), data manipulation instructions (*ADD, SUB*), and data transfer instructions (*MOV, LDA, STA, SWAP*). The descriptions of the instructions are presented in Table 6.2.

Every instruction is represented by a 12-bit data-word. The format of each instruction is described in Table 6.3. Ra, Rb, and Rc are the operand registers, and the #DATA field stores an immediate data value.

The processor instruction set can be extended with more instructions. Instruction size can also be extended beyond the 12-bit data-word. Adding more instructions or extending the size of the instructions will only result in a more featured processor rather than increasing the complexity. In the second case study, this processor is extended with more instructions and a larger number of instruction sizes.

Instruction	Description
NOP	No Operation
ADD Rc, Rb, Ra	Rc $\leftarrow$ (Rb + Rc)
SUB Rc, Rb, Ra	Rc $\leftarrow$ (Rb - Rc)
MOV Rc, Ra	Rc $\leftarrow$ Ra
MOV Rc, #DATA	Rc $\leftarrow$ #DATA
LDA Rc, Ra	Rc $\leftarrow$ @Ra
STA Rc, Ra	@Ra $\leftarrow$ Rc
SWAP Rc, Ra	Rc $\leftarrow$ @Ra $\wedge$ @Ra $\rightarrow$ Rc
JZ #DATA	if ZF then PC $\leftarrow$ #DATA

Tab. 6.2: Description of the instructions

	11	10	9	8	7	6	5	4	3	2	1	0
NOP	0	0	0	X	X	X	X	X	X	X	X	X
ADD	0	0	1	Rc	Rc	Rc	Rb	Rb	Rb	Ra	Ra	Ra
SUB	0	1	0	Rc	Rc	Rc	Rb	Rb	Rb	Ra	Ra	Ra
MOV	0	1	1	0	0	0	Rc	Rc	Rc	Ra	Ra	Ra
MOV	0	1	1	1	#	D	A	T	A	Rc	Rc	Rc
LDA	0	1	1	0	0	1	Rc	Rc	Rc	Ra	Ra	Ra
STA	0	1	1	0	1	0	Rc	Rc	Rc	Ra	Ra	Ra
SWAP	0	1	1	0	1	1	Rc	Rc	Rc	Ra	Ra	Ra
JZ	1	-	#	-	D	-	A	-	T	-	A	-

Tab. 6.3: Op-code of the instructions

Expansion with more instructions and longer instruction sizes only increases the complexity of the design and the verification of the design without really adding anything conceptually new to the system. This expansion will be demonstrated in the second case study within the definition of an ARM7 processor.

Instruction Cycle Operations

Instructions are executed in a variable number of cycles. For example, the data processing instruction is executed in one execution cycle, and the data transfer process uses between two and four cycles. Table 6.4 shows the internal process of an execute cycle. The Address and Data columns represent the value of the address bus and data bus of the processor's external interface. In the address column, ($PC{+}n$) means the address bus has a relative PC value which differs by n. ALU means that the address bus contains the ALU value. In the data-bus column, *[PC+n]*, *[ALU]*, *[Rm]*, and *[Rn]* indicate the memory content with its address indicated by *(PC+n), ALU, Rm*, and *Rn* respectively. The *rw, mreq, seq,* and *lock* denote the value of the signals when an instruction is executed. Flag defines the type of instructions that affect the zero flag.

Instructions	Cycle	Address	Data	rw	mreq	lock	Flag
Data Processing	1	PC+2	[PC+2]	0	0	0	Y
		PC+3					
STA	1	PC+2	[PC+2]	0	0	0	N
	2	ALU	[Rd]	1	0	0	N
		PC+3					
LDA	1	PC+2	[PC+2]	0	0	0	N
	2	ALU	[ALU]	0	1	0	N
	3	PC+3	-	0	0	0	N
		PC+3					
JZ	1	PC+2	[PC+2]	0	0	0	N
	2	ALU	[ALU]	0	0	0	N
	3	ALU+1	[ALU+1]	0	0	0	N
		ALU+2					
SWAP	1	PC+2	[PC+2]	0	0	0	N
	2	Rn	[Rn]	0	0	1	N
	3	Rn	[Rm]	1	1	1	N
	4	PC+3	-	0	0	0	N
		PC+3					

Tab. 6.4: Instruction Cycles

6.2.2 The Formal Specification of a Processor Module

The formal model of the processor is specified in ACL2. In this notation, the formal and mathematical model of the processor is constructed. One can execute and also reason about the model. The processor is modelled in a style suggested by Boyer and Moore [14]. The model used the notion of state which represents the condition of the processor's finite state machine. The state is represented as a linear list. Every component of the state is accessed using an accessor function. Invariants are maintained to ensure that certain relationships hold among the components by using *guard*.

Basic Components

The processor is implemented as a finite state machine at the *Micro-Architecture* (MA) level. Every internal state transition corresponds to a hardware clock cycle. The MA is modelled using a state function, which is a mapping of the form *(f: inputs $\rightarrow$ state $\rightarrow$ state).* The *inputs* argument is the input interface of the processor (*PnReset, PnWait,* and *PDataIn*). The *state* argument defines the internal state of the processor. It contains a list of all components of the processor, such as the registers, flags, etc. The internal registers ($r_0 \ldots r_7$) are represented by variable *preg.* The pipeline registers are denoted by *pp0, pp1*, and *pp2.* Three auxiliary variables are added to the state variable; *pfsm, ptempPC*, and *preset. pfsm* is the processor state machine variable. It indicates the operational state of the processor. *ptempPC* is a temporary PC register. It is used as an intermediate variable to save the next cycle calculation result of PC. *preset* is the variable indicating whether the processor is being reset. In total, a *state* contains thirteen components. A processor's state function is defined as follows:

Definition 7 (ACL2 processor's state: state)

$$\mathit{state}(\mathit{preg\ pp0\ pp1\ pp2\ pzf\ pfsm\ ptempPC\ preset\ plock\ prw\ pmreq\ padd\ pdo})$$
$$\overset{def}{=}$$
$$\mathit{list\ preg\ pp0\ pp1\ pp2\ pzf\ pfsm\ ptempPC\ preset\ plock\ prw\ pmreq\ padd\ pdo}$$

The value of each *state* component can be obtained using accessor functions. For example, function *PReg* is the accessor function for the *preg* component. The definition of *PReg* is shown in Definition 8. In *state*, *preg* is located as the first element of the list. The "*nth(0,s)*" function gets the first element from list s. The remaining accessor functions are as follows: *Pp0, Pp1, Pp2, Pzf, Pfsm, PtempPC, Preset, Plock, Prw, Pmreq, Padd, Pdo.*

Definition 8 (ACL2 accessor function: PReg)

$$\mathit{PReg}(s) \overset{def}{=} \mathit{nth}(0, s)$$

A state predicate *statep* is defined for the state variable *s.* The *statep* function definition is described in Definition 9. The state predicate uses two basic predicates, the natural number predicate (*natnp*) and memory registers predicate (*memp*). Boolean is a primitive type of ACL2. All state components which are of the type boolean do not need to be declared in the statep. Some interpretations of these predicate functions are as follows. The function *(true-listp s)* $\wedge$ *(*$\mid s \mid = $ *13)* means that the state variable *s* has the type list and length 13. The function *(memp(PReg s))* means the component *preg* in state *s* has the type *memp.*

Definition 9 (ACL2 processor state predicate: statep)

$$\begin{aligned} statep(s) \overset{def}{=} & (true - listp\ \ s) \wedge \\ & (\mid s \mid \ = \ 13) \wedge \\ & (memp(PReg\ s)) \wedge \\ & (natnp(Pp0\ s)) \wedge \\ & (natnp(Pp1\ s)) \wedge \\ & (natnp(Pp2\ s)) \wedge \\ & (natnp(Pfsm\ s)) \wedge \\ & (natnp(PtempPC\ s)) \wedge \\ & (natnp(Padd\ s)) \wedge \\ & (natnp(Pdo\ s)) \end{aligned}$$

The function *(natnp x)* restricts the value of x to be a positive integer. Definition 10 describes the natural number predicate function.

Definition 10 (ACL2 natural number predicate: natnp)

$$natnp(x) \overset{def}{=} (x \ \in \ \mathbf{int}) \ \wedge \ (x \ \geq \ 0)$$

The function *(memp x)* defines x as a list of natural numbers. Definition 11 describes the memory predicate.

Definition 11 (ACL2 memory predicate: memp)

$$\begin{aligned} memp(x) \overset{def}{=} & \ if\ (atom\ x) \\ & \quad then\ (x = nil) \\ & \quad else\ (natnp(car\ x)\ \wedge\ memp(cdr\ x)) \end{aligned}$$

In ACL2, predicates are used as guards or invariants. A guard guarantees that certain relationships hold among the components. For example, (*memp mem*) ensures that *mem* is always a list of natural numbers. The state transformer function produces a new state by updating the content of old state components. The state predicate ensures any modification will update only the field and maintain the construct.

The Processor's Instruction Set

Every instruction cycle is modelled as two half-cycle functions, instead of one cycle functional model. In the one cycle functional model, an *oracle* step is needed in the read operation [84]. It is by executing the operation twice. In the first execution, the processor provides the memory address where data is going to be read. In the second execution, the data is read by the processor. The two half-cycle modelling style provides a mechanism to conduct a sequence of simulations between the processor and the memory modules without any *oracle* step. It is also more realistic and closer to the actual operation. All output signals from the processor, including data-output and address, are available in the first half-cycle (cycle$_l$). Data inputs to the processor are handled in the second half-cycle (cycle$_h$). The memory uses the first half cycle information to perform either memory read or memory write operations. The output of the memory module, in a read cycle, is handled by the processor in the second half-cycle. The superscript (1 . . . 4) denotes the cycle number of the instruction. In this section, the model of each instruction will be described.

In general, all first cycle instructions perform a similar task. In the first half-cycle, it increases PC by one and push the pipeline stacks one step forward. The top instruction in the new pipeline stacks is the one being executed. In the second half-cycle, a new instruction fetched by the processor from the memory is stored in the last pipeline stacks.

• **No Operation**: NOP

NOP is a no-operation instruction. The NOP process is completed in a single clock cycle. The *cycle1-nop* function is the first half cycle of NOP operation. The definition of the cycle1-nop function is shown in Definition 12.

Definition 12 (ACL2 first half cycle NOP)
```
(defun cycle1-nop (s)
  (declare (xargs :guard (statep s)))
  (modify s :preg (put 7 (+ 1 (Padd s)) (PReg s))
            :pp0 (Pp1 s)
            :pp1 (Pp2 s)
            :padd (+ 1 (Padd s))
            :prw nil
            :pmreq nil
            :plock nil))
```

The function accepts one argument, the state of processor *s*. The well-formedness of *s* is guaranteed by *(declare (xargs :guard (statep s)))*. This statement is the declaration that the argument *s* is constrained by the state predicate function *statep*. The *modify* function obtains the old values of *s* and updates only the value of components which are modified. In this cycle, seven variables (preg, pp0, pp1, padd, prw, pmreq, pLock) are updated. Let us consider the first update process of Preg. The process uses two accessor functions Padd and Preg to obtain the contents of padd and preg from state *s*. The statement has the following meaning: the value of padd is increased by one and is stored in the register number 7 of preg. The result of this operation is stored back into *s*.

The cycle1-nop function can be represented in mathematical notation as follows:

$$\begin{array}{lllll} NOP_l^1 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{padd+1}) \\ & & \text{pp0} & \leftarrow & \text{pp1} \\ & & \text{pp1} & \leftarrow & \text{pp2} \\ & & \text{padd} & \leftarrow & \text{padd+1} \\ & & \text{prw,pmreq,pLock} & \leftarrow & \text{nil} \end{array}$$

NOP_l^1 is the representation of the first half cycle NOP function. Consider the first line of NOP_l^1 statement, the first variable which is updated is the preg. The first left arrow (preg ←) means the content of preg will be updated. The second arrow ([r7] ←) means the content of register 7 will be updated. (padd + 1) means the value of variable padd is increased by one. (preg ← ([r7] ← padd+1)) means the content of register 7 of variable preg is updated with the value of padd increased by one.

For the remainder of this dissertation, the description of the instruction sets will be presented in this mathematical notation.

$$\begin{array}{lllll} NOP_h^1 & : & \text{pfsm} & \leftarrow & 1 \\ & & \text{pp2} & \leftarrow & \text{di} \\ & & \text{preset} & \leftarrow & \text{t} \end{array}$$

In the second half cycle (NOP_h^1), the variable di is the data-in and the constant t is a high signal.

- **Jump if Zero**: JZ #DATA

$$\begin{array}{lllll} JZ_l^1 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{padd+1}) \\ & & \text{pp0} & \leftarrow & \text{pp1} \\ & & \text{pp1} & \leftarrow & \text{pp2} \\ & & \text{padd} & \leftarrow & \text{padd+1} \\ & & \text{prw,pmreq,plock} & \leftarrow & \text{nil} \\ & & \text{ptempPC} & \leftarrow & \text{\#DATA} \\ JZ_h^1 & : & \text{pfsm} & \leftarrow & \text{ZF} \rightarrow 8 \mid 1 \\ & & \text{pp2} & \leftarrow & \text{di} \\ & & \text{preset} & \leftarrow & \text{t} \end{array}$$

The JZ instruction performs a conditional jump. If ZF is set, then PC is changed. The processor goes into a pipeline flush mode for two cycles to clear the pipeline stacks. During these two cycles, the processor ignores the instruction which is stored in the pipeline. Instead, it executes a process similar to NOP. In this cycle, the new address (#DATA) is stored in ptempPC. If ZF is clear then the processor will perform a process similar to NOP and resume executing the next instruction in the next cycle. In this case the JZ instruction is completed in a single clock cycle.

$$\begin{array}{lllll}
JZ_l^2 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{ptempPC}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{ptempPC} \\
 & & \text{prw,pmreq,plock,pzf} & \leftarrow & \text{nil} \\
JZ_h^2 & : & \text{pfsm} & \leftarrow & 9 \\
 & & \text{pp2} & \leftarrow & \text{di} \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

The second JZ cycle (JZ^2)is only executed whenever the ZF is set. The new address which is stored in ptempPC is assigned to the program counter. The new instruction loaded from external memory is stored in the pipeline. This cycle is also used by other instructions which write the PC register (r_7).

$$\begin{array}{lllll}
JZ_l^3 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{padd+1}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+1} \\
 & & \text{prw,pmreq,plock,pzf} & \leftarrow & \text{nil} \\
JZ_h^3 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{pp2} & \leftarrow & \text{di} \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

The third JZ cycle (JZ^3) is the final pipeline flush cycle. It performs a similar operation as NOP.

- **Addition**: ADD Rc,Rb,Ra

$$\begin{array}{lllll}
ADD_l^1 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{padd+1}) \wedge ([\text{Rc}] \leftarrow [\text{Rb}]+[\text{Ra}]) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+1} \\
 & & \text{prw,pmreq,plock} & \leftarrow & \text{nil} \\
 & & \text{ptempPC} & \leftarrow & [\text{Rc}] \\
 & & \text{pzf} & \leftarrow & ([\text{Rc}]=0) \rightarrow T \ \mid \ nil \\
ADD_h^1 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{pp2} & \leftarrow & \text{di} \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

The ADD instruction adds the value of register *Rb* with the value of register *Ra*. The result is stored in register *Rc*. If the summation result is zero, ZF is set. Otherwise, it is cleared. The ADD operation is completed in a single clock cycle.

- **Subtraction**: SUB Rc,Rb,Ra

SUB_l^1	:	preg	$\leftarrow$	([r7] $\leftarrow$ padd+1) $\wedge$ ([Rc] $\leftarrow$ [Rb]-[Ra])
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+1
		prw,pmreq,plock	$\leftarrow$	nil
		ptempPC	$\leftarrow$	[Rc]
		pzf	$\leftarrow$	([Rc]=0) $\rightarrow T \mid nil$
SUB_h^1	:	pfsm	$\leftarrow$	1
		pp2	$\leftarrow$	di
		preset	$\leftarrow$	t

The SUB instruction subtracts the value of register *Ra* from the value of register *Rb*. The result is stored in register *Rc*. If the subtraction result is zero, ZF is set. Otherwise, it is cleared. The SUB operation is completed in a single clock cycle.

- **Move Data**: MOV Rc,Ra or MOV Rc,#DATA

MOV_l^1	:	preg	$\leftarrow$	([r7] $\leftarrow$ padd+1) $\wedge$
				(([Rc] $\leftarrow$ [Ra]) | ([Rc] $\leftarrow$ #data))
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+1
		prw,pmreq,plock	$\leftarrow$	nil
		ptempPC	$\leftarrow$	[Rc]
		pzf	$\leftarrow$	([Rc]=0) $\rightarrow T \mid nil$
MOV_h^1	:	pfsm	$\leftarrow$	1
		pp2	$\leftarrow$	di
		preset	$\leftarrow$	t

The MOV operation loads the value of register *Ra* or the immediate data *#DATA* and stores it in the register *Rc*. When the transfered data is zero, ZF is set. Otherwise it is cleared. The MOV operation is completed in a single clock cycle.

- **Load Data from External Environment**: LDA Rc,Ra

LDA_l^1	:	preg	$\leftarrow$	([r7] $\leftarrow$ padd+1)
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+1
		prw,pmreq,plock	$\leftarrow$	nil
LDA_h^1	:	pfsm	$\leftarrow$	3
		pp2	$\leftarrow$	di
		preset	$\leftarrow$	t

The LDA instruction loads external data into the processor. The address is defined by the value of register *Ra*. The external data is then stored in register *Rc*. The transfer process is started in the second cycle.

LDA_l^2	:	preg	$\leftarrow$	([r6] $\leftarrow$ padd) $\wedge$ ([r7] $\leftarrow$ [Ra])
		padd	$\leftarrow$	[Ra]
		pmreq	$\leftarrow$	t
		prw,plock	$\leftarrow$	nil
LDA_h^2	:	preg	$\leftarrow$	([Rc] $\leftarrow$ di)
		ppf	$\leftarrow$	0
		pfsm	$\leftarrow$	4
		preset	$\leftarrow$	t

In the second cycle (LDA^2), the PC is updated with the contents of register*Ra*. The old PC value is saved in register r_6. Data from the memory to the processor comes in the second half-cycle. It is stored in register *Rc*.

LDA_l^3	:	preg	$\leftarrow$	([r7] $\leftarrow$ [r6])
		padd,ptempPC	$\leftarrow$	[r6]
		prw,pmreq,plock	$\leftarrow$	nil
LDA_h^3	:	pfsm	$\leftarrow$	1
		preset	$\leftarrow$	t

In the final cycle (LDA^3), the PC is restored to its original value. It is now ready to execute a new instruction in the next clock cycle. The LDA operation is completed in three clock cycles.

- **Store Data in External Environment**: STA Rc,Ra

STA_l^1	:	preg	$\leftarrow$	([r7] $\leftarrow$ padd+1)
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+1
		prw,pmreq,plock	$\leftarrow$	nil
STA_h^1	:	pfsm	$\leftarrow$	2
		pp2	$\leftarrow$	di
		preset	$\leftarrow$	t

The STA instruction gets the data stored in register *Rc* and sends it to the memory. The data is stored externally in the location assigned by the content of register *Ra*. The data transfer process is conducted in the next cycle.

STA_l^2	:	preg	$\leftarrow$	([r6] $\leftarrow$ padd) $\wedge$ ([r7] $\leftarrow$ [Ra])
		padd	$\leftarrow$	[Ra]
		prw	$\leftarrow$	t
		pmreq,plock	$\leftarrow$	nil
		pdo	$\leftarrow$	[Rc]
STA_h^2	:	pfsm	$\leftarrow$	1
		preset	$\leftarrow$	t

In the second cycle (STA^2), the PC is updated with the contents of register*Ra*. The old PC value is saved in register r_6. The processor transfers data from register *Rc* to the data-out port (*pdo*). The PC is restored to the previous address before the next instruction is executed. The STA operation is completed in two clock cycles.

- **Swap Data Inside and Outside of the Processor**: SWAP Rc,Ra

$$\begin{array}{llll}
SWAP_l^1 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow \text{padd}+1) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd}+1 \\
 & & \text{prw,pmreq,plock} & \leftarrow & \text{nil} \\
SWAP_h^1 & : & \text{pfsm} & \leftarrow & 5 \\
 & & \text{pp2} & \leftarrow & \text{di} \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

The *SWAP* instruction exchanges the value of register *Rc* with the contents of the memory location indicated by the content of Register *Ra*. The swapping process involves reading and writing the content of the internal register and the external memory.

$$\begin{array}{lllll}
SWAP_l^2 & : & \text{preg} & \leftarrow & ([\text{r6}] \leftarrow \text{padd}) \wedge ([\text{r7}] \leftarrow [\text{Ra}]) \\
 & & \text{padd} & \leftarrow & [\text{Ra}] \\
 & & \text{prw,pmreq} & \leftarrow & \text{nil} \\
 & & \text{plock} & \leftarrow & \text{t} \\
SWAP_h^2 & : & \text{pfsm} & \leftarrow & 6 \\
 & & \text{ptempPC} & \leftarrow & \text{di} \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

In the second cycle (SWAP^2), the PC is updated with the contents of register*Ra*. The old PC value is saved in register r_6. The incoming external data is stored in the temporary register *ptempPC*.

$$\begin{array}{lllll}
SWAP_l^3 & : & \text{prw,pmreq,plock} & \leftarrow & \text{t} \\
 & & \text{pdo} & \leftarrow & [\text{Rc}] \\
SWAP_h^3 & : & \text{preg} & \leftarrow & ([\text{Rc}] \leftarrow \text{ptempPC}) \\
 & & \text{pfsm} & \leftarrow & 7 \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

In the third cycle (SWAP^3), the contents of register *Rc* is sent to *pdo*. Then the content of register *Rc* is updated with the value of *ptempPC*.

$$\begin{array}{lllll}
SWAP_l^4 & : & \text{preg} & \leftarrow & ([\text{r7}] \leftarrow [\text{r6}]) \\
 & & \text{padd,ptempPC} & \leftarrow & [\text{r6}] \\
 & & \text{prw,pmreq,plock} & \leftarrow & \text{nil} \\
SWAP_h^4 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{preset} & \leftarrow & \text{t}
\end{array}$$

In the final cycle (SWAP^4), the PC is restored to its original value. The SWAP operation is completed in four clock cycles.

The Processor's Execution Model

The top level processor function is the processor's single cycle cyclic behaviour function (*PSCexec*). Similar to the way in which the instruction is modelled, *PSCexec* is split into two half-cycle functions ($PSCexec_l, PSCexec_h$). $PSCexec_l$ is the group of all first half-cycle processes and $PSCexec_h$ is the group of all second half-cycle processes. *PSCexec* takes two group of arguments, input signals (*PnReset, PnWait, PDataIn*) and the current state of the processor (*state*). The result of executing the function with the arguments is an updated *state* of the processor. This input information is also used by $PSCexec_l$ and $PSCexec_h$. $PSCexec_l$ uses *PnReset, PnWait* and *state* whilst $PSCexec_h$ uses all input signals and data. The execution process of *PSCexec* starts with the $PSCexec_l$ and $PSCexec_h$ functions handling the *PnReset* and *PnWait* control signals accordingly. If the control signals instruct the processor to execute the instruction, $PSCexec_l$ and $PSCexec_h$ decode the highest instruction in the pipeline into opcode. Then they perform a case analysis on the opcode and select the appropriate execution function. The description of *PSCexec* is as follows:

Definition 13 (ACL2 processor functional model: PSCexec)
$PSCexec(PnReset\ PnWait\ PDi\ state)$
$\overset{def}{=}$
$PSCexec_h\ PnReset\ PnWait\ PDi\ (PSCexec_l\ PnReset\ PnWait\ state)$

The well-formed condition of the state variable is maintained by showing that an update on the *state* variable maintains its structure. Lemma 1 shows that given a well-formed processor state *s* and a natural number *di*, the result of evaluating *PSCexec* with *state* and *di* maintains the structure of the processor's state.

Lemma 1 (PSCexec-gives-statep)
$((statep\ s)\ \wedge\ (natnp\ PDi)) \rightarrow statep(PSCexec\ PnReset\ PnWait\ PDi\ state)$

Lemma 1 is proved using the well-formedness lemmas of $PSCexec_l$ and $PSCexec_h$. The correctness of both well-formedness lemmas depends on the well-formedness lemmas of functions that construct $PSCexec_l$ and $PSCexec_h$ functions.

The processor (*PSCexec*) is defined using seventy-eight functions, in which forty-one of them update the processor state variable. The well-formedness condition of *state* needs to be preserved by all functions which update the state variable. One lemma, which is similar to Lemma 1, is proven for each of these forty-one functions.

ACL2 proves these lemmas automatically by rewriting. The well-formedness proof of the processor took approximately one minute on a Linux machine with an Intel Pentium II 400MHz processor with 384M RAM.

6.3 Interrupt Model

6.3.1 The Architecture

Interrupt is the second master module. The module implements only the external behaviour of the control signals. The internal functionality of the module is left undefined. This simple model is meant to represent the protocol aspect of a more complex core, for example: a *Digital Signal Processing* (DSP) core, a *Memory Manangement Unit* (MMU), and etc. The module is meant to demonstrate the disruption process on the operation of default master. The behaviour of this module is sufficient for the purpose. The bus-request from the interrupt module can force the bus-controller to stop granting the processor bus-access. One of the consequences is that software execution has to be interrupted and later resumed.

The Interrupt module has two input signals (*InReset, InWait*) and one output signal (*InReadreq*). The description of the signals are as follows:

Name	Description
InReset	is the reset signal. A LOW level input signal forces the interrupt to go to the default reset state.
InWait	is the wait signal. A LOW level input signal stalls the interrupt.
InReadreq	is the request signal. When the signal is LOW, it indicates that the interrupt module requires bus access.

Tab. 6.5: Interrupt Input/Output Interfaces

The behaviour of the module sends a request signal (*InReadreq*) to obtain bus ownership. Whenever granted, it keeps the bus for n cycles before releasing the bus. After m cycles of idle state, the cycle will be repeated.

6.3.2 The Formal Specification of an Interrupt Module

The formal model of an Interrupt module is specified in HOL. The function takes input control signals (*InReset, InWait*) and the state of the interrupt module (*Istate*). *Istate* contains two elements: a counter which defines the number of idle or active cycles of the interrupt and the status of interrupt request (*InReadreq*). The block diagram of an Interrupt finite state machine is described in Figure 6.2

The idle and active cycles are implemented using a 2-bit counter. *InitCounter* initialises the counter to the value of zero. When the counter reaches the maximum value of three, the counter will be re-initialised and the request signal will be toggled. The description of *ISCexec* is provided in Definition 14.

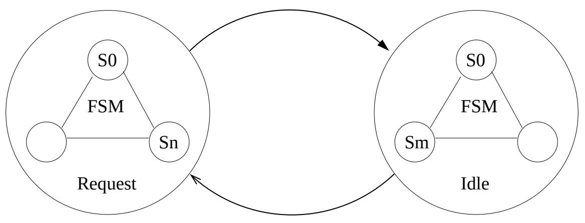

Fig. 6.2: Block Diagram of an Interrupt Finite State Machine

Definition 14 (HOL interrupt functional model: ISCexec)

```
$ISCexec(InReset\ InWait\ Istate)$
$\overset{def}{=}$
$if\ InReset$
$then\ if\ (InReadreq(Istate)\ \vee\ InWait)$
    $then\ if\ (Counter(Istate) = n)$
        $then\ (InitCounter, \neg(InReadreq(Istate)))$
        $else\ inc\_counter(Istate)$
    $else\ Istate$
$else\ (InitCounter, InReadreq)$
```

6.4 Memory Model

Memory is the only slave module in the SIP architecture. It is specified to be capable of responding to any request during one clock cycle. It is also assumed that the module covers all address spaces.

The module contains two types of memory, *Read Only Memory* (ROM) and *Random Access Memory* (RAM). An application in the form of program codes is stored in the ROM. In this way, we prevent it from being a self-modifying system. All other data used in the operations are stored in the RAM component.

6.4.1 The Architecture

The memory module has five external interfaces, four inputs and one output. The inputs are (*MnReset, MnRW, MAddress, MDataIn*) and the output is (*MDataOut*). A description of these input and output signals is shown in Table 6.6

Name	Description
MnReset	is the reset signal. A LOW level input signal forces the memory to go to its default reset state.
MnRW	is the input read or write signal. Memory is in the read cycle when the signal is LOW. Otherwise, it is in the write cycle.
MAddress	is the address line. This input line provides the memory address for the memory module to operate.
MDataIn	is the input data line. This input line allows the data to be written into the memory module in the write cycle.
MDataOut	is the output data line. This output line provides the data to be transfered out of the memory module.

Tab. 6.6: Memory Input/Output Interfaces

6.4.2 The Formal Specification of a Memory Module

A memory module is formally specified in ACL2. Modelling the entire memory as an array may consume a large amount of space and make the approach unrealistic. Another approach is to build a model where only the necessary elements exist in the model [60, 68, 82]. In this approach, a memory element is represented as a tuple *(add, val)*. The tuple represents the content of memory *val* as stored in the address *add*. The description of a memory-cell is provided in Definition 15.

Definition 15 (ACL2 memory cell: MemCell)
$MemCell(address, value) \overset{def}{=} (address, value)$

A memory module is represented as a list of memory-cells. Initially, the module has a default value of an empty list. Every time the module is written, either a memory cell is added to the list or the existing memory-cell is updated with a new value. This representation removes the need for representing the whole array of memory-cells.

We define two predicates for the memory cell model: *MemCellp* and *MemModulep*. *(MemCellp x)* ensures that x is a memory cell which contains two natural numbers: an address identifier and its contents. *(MemModulep x)* guarantees that x is a memory module which is a list of memory cells. Descriptions of *MemCellp* and *MemModulep* are provided in Definition 16 and Definition 17 respectively.

Definition 16 (ACL2 memory cell predicate: MemCellp)
$MemCellp(Address, Value) \overset{def}{=} ((natnp\ address)\ \wedge\ (natnp\ value))$

Definition 17 (ACL2 memory module predicate: MemModulep)

```
$MemModulep(mem) \overset{def}{=}$ $if\ (atom\ mem)$
                           $then\ (mem = nil)$
                           $else\ (MemCellp(car\ mem)\ \wedge\ MemModulep(cdr\ mem))$
```

Similarly to the processor module, the memory module is also defined using a state function ($f : inputs \rightarrow state \rightarrow state$). The inputs are the input interface of the memory (*MnReset, MnRW, MAddress, MDataIn*). The state is the internal state of the memory. A memory state variable contains two variables: a list of memory cells and output data. The first element is a list of MemCell. The second element is a natural number and represents the data on the output port of the memory module. The definition for a memory state variable is as follows:

Definition 18 (ACL2 memory's state: MemState)

$MemState(mlist\ mdo) \overset{def}{=} list\ mlist\ mdo$

The behaviour of a memory module is formalised by the *MSCexec* function. It takes five arguments, *MnReset, MnRW, MDi, MAdd* and *Mstate*. The *nMreset* sets the memory module into an initial state *memReset*. The initial condition of the memory module can be defined in various forms. It can be in the form of an empty list or contain a list of memory cell values. A pre-loaded application code can be represented in this form. The *MnRW* defines the memory read or write operation. A LOW signal of *MnRW* means that the memory module is performing a read operation, while a HIGH signal means a write operation. The description of *MSCexec* function is as follows:

Definition 19 (ACL2 memory functional model: MSCexec)

```
$MSCexec(MnReset\ MnRW\ MDi\ MAdd\ Mstate)$
$\overset{def}{=}$
$if\ MnReset$
$then\ if\ MnRW$
    $then\ (WriteMemSt\ MAdd\ MDi\ Mstate)$
    $else\ (ReadMemSt\ MAdd\ Mstate)$
$else\ memReset$
```

Similar to the processor, Lemma 2 shows that a given *Mstate* satisfies the memory state predicate and both address and data-in are natural numbers, the result of evaluating *MSCexec* with *Mstate*, *MDi*, and *MAdd* satisfies *MemStatep.*

Lemma 2 (MemExec-gives-memstatep)

$((MemStatep\ Mstate)\ \wedge\ (natnp\ MDi)\ \wedge\ (natnp\ MAdd))$

$\rightarrow$

$MemStatep(MSCexec\ MnReset\ MnRW\ MDi\ MAdd\ Mstate)$

Lemma 2 depends on the well-formed results of the memory write operation (*WriteMemSt*) and the memory read operation (*ReadMemSt*). ACL2 proves these three lemmas automatically by rewriting.

6.4.3 Application Code

Embedded software is a vital component in SoC design. It is used to derive a series of product using the same platform. The embedded software component is stored in the memory module. In general, there are two ways to have the embedded software in the memory module. The first one is by having the embedded software as part of the memory module, such as storing the code in the ROM part of the memory module. The second one is by uploading the embedded software into the memory module when the system is started. In this case study, the first approach is used. The embedded software is stored in the ROM part of the memory module.

Each instruction of the embedded software occupies one memory cell. It is defined as a tuple of address and code. The address is the memory location in which the instruction is stored. The code is the instruction representation in machine code. For example, the instruction (MOV R_5,#11) stored in memory location 3 is represented by (memcell 3 1933). Instructions of the embedded software in the memory module are stored in the memory module and is defined as a list of memory cells.

A fragment of embedded software code is implemented in SIP as part of the memory module. This code will be used to demonstrate the verification of a SIP hardware/software system. The code is stored in the ROM which prevents the system from being able to modify the code. It performs an addition operation, which involves reading the operand values from the memory and writing back the result. It is described as follows:

Address	Instructions		Code	Memory Element
	...			
1	MOV	R_5,#10	1925	(memcell 1 1925)
2	LDA	R_0,R_5	1640	(memcell 2 1640)
3	MOV	R_5,#11	1933	(memcell 3 1933)
4	LDA	R_1,R_5	1641	(memcell 4 1641)
5	ADD	R_2,R_1,R_0	648	(memcell 5 648)
6	MOV	R_5,#12	1941	(memcell 6 1941)
7	STA	R_5,R_2	1706	(memcell 7 1706)
	...			

The above instructions are translated into ACL2 as a list. The order of memory cells in the list is not important, provided all data are using unique memory addresses. If there are two or more memory cells which have the same address, the most recent memory cell will be used in the read/write operations. The addition operation codes are described as follows:

Definition 20 (ACL2 Memory Initial Condition: memResetLst)

```
(defun memResetLst nil
   (declare (xargs :guard t))
   (list (memcell 1 1925)
         (memcell 2 1640)
         (memcell 3 1933)
         (memcell 4 1641)
         (memcell 5 648)
         (memcell 6 1941)
         (memcell 7 1706)))
```

The memory module uses the above function to initialise the module. Whenever a reset signal is fed to the module, it automatically deletes all data in the list and replaces them with the contents of ROM (*memResetLst*).

6.5 System Integration Module and Bus Control (SIMBC)

All master and slave modules are integrated through a common bus system. *System Integration Module and Bus Control* (SIMBC) is the platform where modules are integrated. SIMBC contains two main components, the arbiter and the bus multiplexer. The arbiter manages the control signals, while the bus multiplexer provides the mechanism for data transfers.

6.5.1 The Arbiter

The arbiter is responsible for managing the control of data transactions between modules. It responds to request signals from masters by granting one of them access to the data-bus and providing read or write signals to the memory module. The arbiter implements a bus priority arbitration scheme. The interrupt has the highest priority, while the processor gets the lowest priority. A request from the interrupt is always granted, except when the processor is in lock mode. When no master is requesting to access the bus, the default master/processor is granted bus access. A SIMBC state transition diagram is presented in Figure 6.3

In general, the arbiter module can be implemented in any of the supported systems such as HOL, ACL2, and SMV. One consideration is that an arbiter is

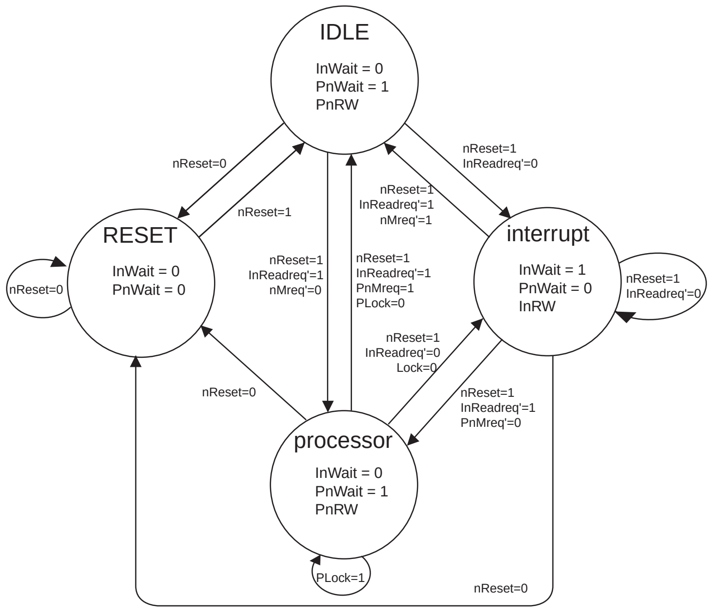

Fig. 6.3: SIMBC State Transitions Diagram.

normally verified using a model checker. A preferable approach is to describe it in a system which can perform model checking. The other approach is to describe the arbiter as a collection of properties about the behaviour of its interfaces. In this approach, attention is focused on defining the intended properties. The properties are the golden model in describing the arbiter implementation.

In this case study, the arbiter module is specified as a set of LTL specifications. It is an abstraction which represents the external interface which is expected to be found in any implementation. The specifications describe the input/output relational behaviours of the module. The specifications of the arbiter are described in Definition 21.

Definition 21 (HOL Arbiter)

Arbiter *SnReset SPnWait SPLock SPnRW SInReadreq SInWait SInRW SMnRW*

$$\overset{def}{=}$$

$$
\begin{aligned}
&G((\neg SPnWait \ \lor \ \neg SInWait) \ \land \\
&\quad (SPnWait \ \rightarrow \ (SMnRW = SPnRW)) \ \land \\
&\quad (SInWait \ \rightarrow \ (SMnRW = SInRW)) \ \land \\
&\quad ((\neg SPnWait \ \land \ \neg SInWait) \rightarrow \neg SMnRW) \ \land \\
&\quad (SnReset \rightarrow (SPnWait \ \lor \ SInWait)) \ \land \\
&\quad (\neg SnReset \rightarrow (\neg SPnWait \ \land \ \neg SInWait)) \ \land \\
&\quad (\neg SnReset \rightarrow (X \ SnReset \rightarrow X \ SPnWait)) \ \land \\
&\quad ((X \ SnReset \ \land \ SPLock) \rightarrow X \ SPnWait) \ \land \\
&\quad ((X \ SnReset \ \land \ SInReadreq) \rightarrow X \ SPnWait) \ \land \\
&\quad ((X \ SnReset \ \land \ \neg SPLock \ \land \ \neg SInReadreq) \rightarrow X \ SInWait)
\end{aligned}
$$

The later case study presented in Chapter 8 and 9 will describe in more detail the approach to obtain the properties from a specific implementation.

6.5.2 The Bus Multiplexer

The bus multiplexer is responsible for managing the selection of which module's data are available in the data-bus and address-bus. The selection depends on the output control signals from the arbiter. Similar to the arbiter, the bus multiplexer module can be implemented in any of the supported systems such as HOL, ACL2, and SMV. For the same reason, the bus multiplexer module is specified using LTL specifications. The specification describes the input/output relational behaviour of the module. When the processor is granted the bus, the address-bus value comes from the processor and the data-bus comes either from the processor or the memory. A similar behaviour is also applied to the interrupt module. The specification of the bus module is described in Definition 22.

Definition 22 (HOL Bus-Mux)

$$\begin{array}{l}
\mathbf{Bus-Mux}\ \mathit{SPnWait}\ \mathit{SPnRW}\ \mathit{SPDi}\ \mathit{SPDo}\ \mathit{SPAdd}\ \mathit{SInWait}\ \mathit{SInRW}\ \mathit{SIDi}\ \mathit{SIDo}\ \mathit{SIAdd}\\
\qquad\qquad \mathit{SMDi}\ \mathit{SMDo}\ \mathit{SMAdd}\\
\overset{def}{=}\\
G(\mathit{SPnWait} \rightarrow ((\mathit{SPnRW} \rightarrow ((\mathit{SPDo} = \mathit{SMDi})\ \wedge\ (\mathit{SPDi} = \mathit{SPDo})))\ \wedge\\
\qquad\qquad (\neg\mathit{SPnRW} \rightarrow ((\mathit{SMDo} = \mathit{SMDi})\ \wedge\ (\mathit{SPDi} = \mathit{SMDo})))\ \wedge\\
\qquad\qquad (\mathit{SPAdd} = \mathit{SMAdd})))\ \wedge\\
G(\mathit{SInWait} \rightarrow ((\mathit{SInRW} \rightarrow ((\mathit{SIDo} = \mathit{SMDi})\ \wedge\ (\mathit{SIDi} = \mathit{SIDo})))\ \wedge\\
\qquad\qquad (\neg\mathit{SInRW} \rightarrow ((\mathit{SMDo} = \mathit{SMDi})\ \wedge\ (\mathit{SIDi} = \mathit{SMDo})))\ \wedge\\
\qquad\qquad (\mathit{SIAdd} = \mathit{SMAdd})))
\end{array}$$

6.6 The Verification Platform

In the verification platform, modules are connected and integrated in the HOL theorem prover environment. Every module which is not formalised in HOL needs to have an interface to bridge their differences in formalism. The formal verification environment described in Chapter 3 has a collection of methods and techniques to create such links or interfaces. For example: the PROSPER and ACL2PII framework provides the environment to define connections between ACL2 and HOL. The ACL2 formal model can be interpreted in HOL automatically. In this section, the construction of the SIP formal verification platform will be described.

6.6.1 Interfacing Processor-Module

Sharing knowledge between ACL2 and HOL means both systems must have an identical interpretation of the same piece of information. This is achieved by defining the relationship between definitions in ACL2 and HOL. It is sufficient to have a middle level of abstraction linking these two definitions. For example, the interface allows an ACL2 function definition to be imported into HOL. On the other hand, a HOL constant representation is sufficient to represent an ACL2 function in HOL.

The internal representation of the processor is defined by *state*. In ACL2, the *state* is represented as a list of variables. The type of the list is defined by the state predicate function *statep*. In HOL, these two definitions are joined together and used in defining the structure of *Pstate_ty*. *Pstate_ty* is the type of processor state in HOL. The definition of the type structure is given in Definition 23.

Definition 23 (Type: Pstate_ty)

Pstate_ty
$\overset{def}{=}$
: *num list#num#num#num#bool#num#num#bool#bool#bool#bool#bool#num#num*

Seven HOL constants are declared to represent ACL2 functions. The constants are the processor execution function (*PSCexec*), the state predicate (*statep*), memory request (*PnMreq*), read/write request (*PnRW*), lock request (*PLock*), data-out (*PDo*), and address (*PAdd*). The last four constants are also a group of accessor functions. Constants are created in HOL using the *mkbasefun* function. The *mkbasefun* function takes four arguments. The first and third arguments represent the ACL2 function name and the HOL constant. The second argument is the number of inputs for the functions. The last argument is the type definition for the constant. For example, the HOL constant for the memory request function is defined as (mkbasefun “PnMreq” 1 “PNMREQ” Pstate_ty→bool). By correlating *PnMreq* and *PNMREQ*, the system recognises the ACL2 function PnMreq to have the same meaning as the uninterpreted HOL constant *PNMREQ*. The seven HOL constants are described in Table 6.7.

In HOL, we transform the *PSCexec* functional model into *SPC*. *SPC* is a relational model representation of *PSCexec*. The model uses time-varying inputs and outputs. It employs the usual notation of signal, which is a Boolean valued function taking discrete time arguments. An initial state of the processor (Pst_0) is supplied for the base case (t=0). The result of executing the *PSCexec* function at time t is defined as the next state value (SPst.(t+1)).

ACL2 name	HOL constant	HOL type
PSCexec	PSCEXEC	bool→bool→num→Pstate_ty→Pstate_ty
PSCexec$_l$	PSCEXEC$_l$	bool→bool→Pstate_ty→Pstate_ty
PSCexec$_h$	PSCEXEC$_h$	bool→bool→num→Pstate_ty→Pstate_ty
statep	STATEP	Pstate_ty→bool
PnMreq	PNMREQ	Pstate_ty→bool
PnRW	PNRW	Pstate_ty→bool
PLock	PLOCK	Pstate_ty→bool
PDo	PDO	Pstate_ty→num
PAdd	PADD	Pstate_ty→num

Tab. 6.7: HOL abstract representations of ACL2 Processor functions

Definition 24 (HOL processor relational model: SPC)
$SPC\ SPnReset\ SPnWait\ SPDi\ SPst\ Pst_0$
$\overset{def}{=}$
$(SPst\ 0 = Pst_0) \land$
$\forall t.\ (SPst(t+1) = PSCEXEC(SPnReset\ t)(SPnWait\ t)(SPDi\ t)(SPst\ t))$

The well-formed nature of the processor state has to be maintained by the *SPC* function. The well-formedness lemma is shown in Lemma 3. The lemma describes that given the initial state of the processor Pst_0 satisfies the state-predicate *STATEP*, data-in *di* is always a natural number, and the state of *SPst* at (t+1) is obtained by evaluating *PSCEXEC* function with the state of *SPreset, SPwait, SPdi,* and *SPst* at t, then *SPst* will always satisfy the state-predicate *STATEP*.

Lemma 3 (SPCGivesStatep_H)
$SPC\ SPnReset\ SPnWait\ SPDi\ SPst\ Pst_0 \land$
$STATEP\ Pst_0 \land$
$\forall t.\ natnp(SPDi\ t)$
$\rightarrow$
$\forall t.\ STATEP(SPst\ t)$

The HOL execution code to prove Lemma 3 is presented as follows:

```
- val PSCexecGivesStatep_A = GEN_ALL (getthm [] "PSCEXEC-GIVES-STATEP");
> val PSCexecGivesStatep_A =
      |- !WT RST S DI.
         NATNP DI /\ STATEP S ==> STATEP (PSCEXEC RST WT DI S) : thm
- val SPCGivesStatep_H = TAC_PROOF(([],--`
    !SnReset SPnWait SPDi PS0 SPState.
    (SPC SnReset SPnWait SPDi PS0 SPState /\
     (!t. NATNP (SPDi t)) /\ STATEP PS0)
    ==>
    (!t. STATEP (SPState t))`--),
    ... THEN (Induct_on `t`)
    ... THEN (FULL_SIMP_TAC hol_ss [PSCexecGivesStatep_A]));
```

```
> val SPCGivesStatep_H =
      |- !SnReset SPnWait SPDi PS0 SPState.
         SPC SnReset SPnWait SPDi PS0 SPState /\ (!t. NATNP (SPDi t)) /\
         STATEP PS0 ==>
         !t. STATEP (SPState t) : thm
```

The proof of Lemma 3 depends on *PSCexec-gives-statep* lemma (Lemma 1). The ACL2 lemma is imported into HOL using the *getthm* command. The HOL version of Lemma 1 is stored as *PSCexecGivesStatep_A*. Lemma 3 is then proved by induction on t and using *PSCexecGivesStatep_A*.

Finally, we define the processor module with its input and output interfaces. The output signals (*SPnMreq, SPLock, SPAdd, SPdo, SPnRW*) are obtained by applying the accessor functions to its state variable (*SPstate*). The processor module description is provided in Definition 25.

Definition 25 (HOL processor core: Processor)

$\textbf{Processor}\ \mathit{SPnReset\ SPnWait\ SPDi\ PS_0\ SPstate\ SPnMreq\ SPLock\ SPnRW}$
$\qquad \mathit{SPDo\ SPAdd}$
$\overset{def}{=}$
$\mathit{SPC\ SPnReset\ SPnWait\ SPDi\ SPstate\ PS_0} \wedge$
$\forall t.\ (\mathit{SPnMeq}\ t) = \mathit{PNMREQ}(\mathit{SPstate}(t+1)) \wedge$
$\forall t.\ (\mathit{SPLock}\ t) = \mathit{PLOCK}(\mathit{SPstate}(t+1)) \wedge$
$\forall t.\ (\mathit{SPnRW}\ t) = \mathit{PNRW}(\mathit{SPstate}(t+1)) \wedge$
$\forall t.\ (\mathit{SPDo}\ t) = \mathit{PDO}(\mathit{SPstate}(t+1)) \wedge$
$\forall t.\ (\mathit{SPAdd}\ t) = \mathit{PADD}(\mathit{SPstate}(t+1)) \wedge$
$\forall t.\ \mathit{NATNP}(\mathit{SPDi}\ t) \wedge$
$\mathit{STATEP}(\mathit{Pst_0})$

6.6.2 Interfacing Interrupt-Module

The interrupt module is already modelled in HOL. It transforms a direct representation of the functional model into a relational one. The transformation uses a similar approach as the processor module. The relational model of the interrupt module is implemented as the base case and the step case. In the base case, the initial state of the module (Ist$_0$) is defined. The step case is the next state function for the interrupt module. The relational definition of *ISCexec* is described as follows:

Definition 26 (HOL interface module: SIC)

$\mathit{SIC\ SInReset\ SInWait\ SIst\ Ist_0}$
$\overset{def}{=}$
$(\mathit{SIst}\ 0 = \mathit{Ist_0}) \wedge$
$\forall t.\ (\mathit{SIst}(t+1) = \mathit{ISCexec}(\mathit{SInReset}\ t)(\mathit{SInWait}\ t)(\mathit{SIst}\ t))$

The Interrupt module is constructed from SIC and its state values. Three state signals are obtained from the interrupt state variable. SIreq is the interrupt request signal, SI_0 and SI_1 are the counters that define how long the module is in active and idle mode. One output signal and three bus interfaces are introduced. Four signal and bus uninterpreted interfaces are introduced as part of interrupt module interfaces. These interfaces represent input/output connections of the module without internal functionality being implemented. These interfaces are one read/write output signal (SInRW) and three buses: the data-input bus (SIDi), the address bus (SIAdd), and the data-output bus (SIDo). The Interrupt module is defined in Definition 27.

Definition 27 (HOL interrupt core: Interrupt)
Interrupt *SInReset SInReadreq SInWait SInRW SIDi SIAdd SIDo* SI_0 SI_1 *SIst* Ist_0
$\overset{def}{=}$
$SIC\ SInReset\ SInWait\ SIst\ Ist_0 x \wedge$
$\forall t.\ (SInReadreq\ t) = SND(SIst(t+1)) \wedge$
$\forall t.\ (SI_0\ t) = FST(FST(SIst(t+1))) \wedge$
$\forall t.\ (SI_1\ t) = SND(FST(SIst(t+1)))$

6.6.3 Memory-Module

Similarly to the processor, the memory is modelled in ACL2. The state representation of the memory is described in HOL. Two memory state representations are defined in Definition 28 and 29; the memory structure (*MemModule_ty*) which contains the tuples of (address,value) and the memory component structure which contains the tuples (address,value) and the data-out line or register of the memory.

Definition 28 (Type: MemModule_ty)
$MemModule\_ty \overset{def}{=} (num\#num)list$

Definition 29 (Type: Mstate_ty)
$Mstate\_ty \overset{def}{=} (num\#num)list\#num$

Thirteen HOL constants are defined to represent ACL2's abstract functions. The functions can be grouped into functions for data constructors, data accessors, data processing, and data predicates. The data constructor functions are the memory cell constructor (*MemCell*) and the memory state constructor (*MemState*). The data accessor functions are the content accessor of a memory cell (*GetMemCell*), the memory cells accessor (*MemList*), and the output data register accessor (*MemDo*). The data processing functions are the memory execution function (*MEMexec*), the memory write function (*MEMexecW*), the memory read function (*MEMexecR*), and the test function (*IsMemExists*). Finally, the predicate functions for memory cell (*MemCellP*), memory cells (*MemP*), memory module (*MemModuleP*), and memory state (*MemStateP*). The thirteen HOL memory representations are described in Table 6.8.

ACL2 name	HOL constant	HOL type
MSCexec	MSCEXEC	bool→bool→num→num→Mstate_ty→ Mstate_ty
MEMexecW	MEMEXECW	bool→num→num→MemModule_ty→ MemModule_ty
MEMexecR	MEMEXECR	bool→num→MemModule_ty→num
MemList	MEMLIST	Mstate_ty→MemModule_ty
MemDo	MEMDO	Mstate_ty→num
MemState	MEMSTATE	MemModule_ty→num→Mstate_ty
MemStateP	MEMSTATEP	Mstate_ty→bool
MemP	MEMP	num list→bool
MemCell	MEMCELL	num→num→MemCell_ty
MemCellP	MEMCELLP	MemCell_ty→bool
MemModuleP	MEMMODULEP	MemModule_ty→bool
IsMemExists	ISMEMEXISTS	num→MemModule_ty→bool
GetMemCell	GETMEMCELL	num→MemModule_ty→MemCell_ty

Tab. 6.8: HOL abstract representations of ACL2 functions

In HOL, we transformed the *MEMexec* functional model into *SMC*. *SMC* is the relational representation of *MEMexec*. The *ISMEMEXISTS* function is the memory decoder which partitions the memory module into several memory components. The initial condition of *SMC* is defined by Mst_0. If *ISMEMEXISTS* is TRUE then the result of executing a *MEMEXEC* function at time t is defined as the next state value (SMst.(t+1)). If it is FALSE then the internal state of the memory remains the same. A description of *SMC* is provided in Definition 30.

Similarly to the processor module, the well-formed nature of the memory state *SMst* has to be maintained. The proof is performed using induction on t and the HOL version of Lemma 2.

Definition 30 (HOL memory module: SMC)

$$
\begin{array}{l}
\mathit{SMC\ SMnReset\ SMDi\ SMAdd\ SMnRW\ SMst\ SMDo\ SMst\ Mst_0} \\
\overset{def}{=} \\
(\mathit{SMst}\ 0 = \mathit{Mst_0}) \wedge \\
\forall t.\ \mathit{SMst}(t+1) = \\
\quad (\mathit{if}\ (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst}\ t))) \\
\quad\ \mathit{then}\ (\mathit{MSCEXEC}(\mathit{SMnReset}\ t)(\mathit{SMnRW}\ t)(\mathit{SMDi}\ t)(\mathit{SMAdd}\ t)(\mathit{SMst}\ t)) \\
\quad\ \mathit{else}\ (\mathit{MEMSTATE}(\mathit{MEMLIST}(\mathit{SMst}\ t))(\mathit{SMDo}\ t))) \wedge \\
\forall t.\ \mathit{NATNP}(\mathit{SMDi}\ t) \wedge \\
\forall t.\ \mathit{NATNP}(\mathit{SMAdd}\ t) \wedge \\
\forall t.\ \mathit{NATNP}(\mathit{SMDo}\ t) \wedge \\
\mathit{MEMSTATEP}(\mathit{Mst_0})
\end{array}
$$

In SIP, the memory module is constructed from two memory components. Every memory address has a unique location within the memory module and all memory components share the data output register. These rules are described in Definition 31.

Definition 31 (HOL memory module constraints: MemRules)

$$
\begin{array}{l}
\mathit{MemRules\ SMAdd\ SMst\ SMst_1\ SMst_2\ Mst_0} \\
\overset{def}{=} \\
\forall \mathit{add}\ t.\ \mathit{GETMEMCELL}(\mathit{SMAdd}\ t)(\mathit{MEMELIST}(\mathit{SMst}\ t)) = \\
\qquad\quad (\mathit{if}\ (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_1}\ t))) \\
\qquad\quad\ \mathit{then}\ (\mathit{GETMEMCELL}(\mathit{SMAdd}\ t)(\mathit{MEMELIST}(\mathit{SMst_1}\ t))) \\
\qquad\quad\ \ \mathit{else}\ (\mathit{GETMEMCELL}(\mathit{SMAdd}\ t)(\mathit{MEMELIST}(\mathit{SMst_2}\ t)))) \wedge \\
\forall \mathit{add}\ t.\ (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_1}\ t)) \vee \\
\qquad\quad \mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_2}\ t))) \wedge \\
\forall \mathit{add}\ t.\ (\neg(\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_1}\ t))) \vee \\
\qquad\quad \neg(\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_2}\ t)))) \wedge \\
\forall t.\ \mathit{MEMDO}(\mathit{SMst}\ t) = \\
\quad (\mathit{if}\ (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMst_1}\ t))) \\
\quad\ \mathit{then}\ (\mathit{MEMDO}(\mathit{SMst_1}\ t)) \\
\quad\ \mathit{else}\ (\mathit{MEMDO}(\mathit{SMst_2}\ t))) \wedge \\
\mathit{MEMSTATEP}(\mathit{Mst_0})
\end{array}
$$

Finally, we define the memory module (*memory*) with its input and output interfaces. The module contains two memory elements defined by their state variables (SMst_1, SMst_2). The outputs of this memory module (*SMDo, SMlist*) are obtained by applying the accessor functions to its state variable (*SMst*). The memory module is defined in Definition 32.

Definition 32 (HOL memory core: Memory)

Memory $\mathit{SMnReset}\ \mathit{SMnRW}\ \mathit{SMDi}\ \mathit{SMAdd}\ \mathit{SMDo}\ \mathit{SMlist}\ \mathit{SMst}\ \mathit{SMst}_1\ \mathit{SMst}_2$
$\qquad \mathit{Mst}_0\ \mathit{Mst}_1\ \mathit{Mst}_2$

$\overset{def}{=}$

$\mathit{SMC}\ \mathit{SMnReset}\ \mathit{SMDi}\ \mathit{SMAdd}\ \mathit{SMnRW}\ \mathit{SMst}_1\ \mathit{SMDo}\ \mathit{SMst}\ \mathit{Mst}_1\ \wedge$

$\mathit{SMC}\ \mathit{SMnReset}\ \mathit{SMDi}\ \mathit{SMAdd}\ \mathit{SMnRW}\ \mathit{SMst}_2\ \mathit{SMDo}\ \mathit{SMst}\ \mathit{Mst}_2\ \wedge$

$\mathit{MemRules}\ \mathit{SMadd}\ \mathit{SMst}\ \mathit{SMst}_1\ \mathit{SMst}_2\ \mathit{Mst}_0\ \wedge$

$\forall t.\ (\mathit{SMDo}\ t = \mathit{MEMDO}(\mathit{SMst}\ t))\ \wedge$

$\forall t.\ (\mathit{SMlist}\ t = \mathit{MEMLIST}(\mathit{SMst}\ t))$

6.6.4 SIP

The SIP platform is implemented by integrating the processor module, interrupt module, memory module, and SIMBC module. The architecture of the platform is presented in Figure 6.4. Interconnection of the components in the verification platform is straightforward. The formal models are integrated and connected using higher order logic to compose relational predicates that model each component. All nodes which share the same connection are defined with the same name. The SIP is defined below:

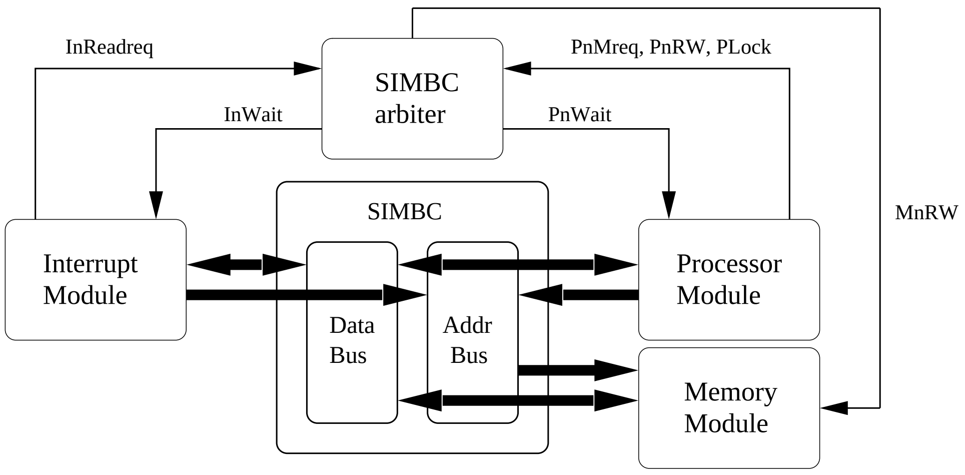

Fig. 6.4: Simple Integration Platform

Definition 33 (HOL the verification platform: SIP)

$$
\begin{aligned}
&\textbf{SIP}\ \mathit{SnReset\ SPnWait\ SPnMreq\ SPDi\ SPLock\ SPAdd\ SPDo\ SPnRW\ SPst\ Pst_0}\\
&\quad \mathit{SInReadreq\ SInWait\ SInRW\ SIDi\ SIAdd\ SIDo\ SI_0\ SI_1\ SIst\ Ist_0}\\
&\quad \mathit{SMnRW\ SMDi\ SMAdd\ SMDo\ SMlist\ SMst\ SMst_1\ SMst_2\ Mst_0\ Mst_1\ Mst_2}\\
&\overset{def}{=}\\
&\textbf{Processor}\ \mathit{SnReset\ SPnWait\ SPDi\ PS_0\ SPState\ SPnMreq\ SPLock\ SPnRW}\\
&\qquad \mathit{SPDo\ SPAdd}\ \wedge\\
&\textbf{Interrupt}\ \mathit{SnReset\ SInReadreq\ SInWait\ SInRW\ SIDi\ SIAdd\ SIDo\ SI_0\ SI_1\ SIst\ Ist_0}\ \wedge\\
&\textbf{Memory}\ \mathit{SnReset\ SMnRW\ SMDi\ SMAdd\ SMDo\ SMlist\ SMst\ SMst_1\ SMst_2\ Mst_0}\\
&\qquad \mathit{Mst_1\ Mst_2}\ \wedge\\
&\textbf{Arbiter}\ \mathit{SnReset\ SPnWait\ SPLock\ SPRW\ SInReadreq\ SInWait\ SMnRW}\ \wedge\\
&\textbf{Bus}-\textbf{Mux}\ \mathit{SPnWait\ SPRW\ SPDi\ SPDo\ SPAdd\ SInWait\ SIRW\ SIDi\ SIDo\ SIAdd}\\
&\qquad \mathit{SMDi\ SMDo\ SMAdd}
\end{aligned}
$$

6.7 Summary

This chapter described the specifications and formalisms of SIP. SIP is a case study used to demonstrate the development of a formal verification platform in a heterogenous formal verification environment. The decision on how to formalise the design is based on two components: what modules are used to build the SoC design and what properties will be verified. Each module of the platform is formalised in the most suitable form for verification.

The first stage in defining how to formalise the design is gathering the design and target verification information. First, SIP is a platform which contains four modules: processor module, interrupt module, memory module, and SIMBC module. Second, two properties of SIP are verified. The first property is the liveness of bus request signal from the master modules. The second property is the correctness of embedded software when it is evaluated in SIP. In the second stage, the decision how to formalised each module is taken based on these two properties. The processor, the memory and the embedded software are formalised in the symbolic simulation environment (ACL2). The SIMBC is the main module in the liveness verification. It is formalised in the model checking environment (SMV). The interrupt module is formalised in the theorem proving environment (HOL). All modules are then integrated in the HOL environment.

Integrating modules with different formalisms require a layer of interface that defines the relationship between different formalisms. HOL is used as the environment to integrate the modules. Any module which is not implemented in HOL has to have logical interfaces which create the relationship between definitions in both systems. The logical connection of SMV is provided by embedding LTL in HOL. An interface is created for each module defined in ACL2. The interface contains the ACL2 data-types correlation in an HOL structure and ACL2 functions are represented as HOL constants.

The formal verification platform of SIP is defined by integrating the modules in higher order logic. This is achieved by connecting the relational predicates of each module. All modules defined in the functional modelling style need to be transformed to the relational modelling style. In this approach, every node which shares the same connection also shares the same connection name.

Chapter 7

The Formal Verification of Simple Integration Platform

As described in Section 3.5, the SoC design validation is based on three validation steps: validation at the block level, interface level, and full-chip level. The SIP validations use the last two steps. This chapter is intended to demonstrate the feasibility of a top level verification of the whole system. It is assumed that all SIP components are correct. Subsequently, no block level validations are performed for SIP. The functional correctness in block level validation is replaced by validating the component to obtain its implementation properties. In the interface level validation, the goal is to validate the liveness properties of the system. Finally, the full-chip level validation is explained by describing the correct execution of a software application that is embedded in the system.

7.1 SIP Liveness Proofs

Every master that wants to access the bus has to send a request signal to the arbiter. The arbiter processes the requests using a fixed priority arbitration scheme and decides which master is granted access to the bus. The liveness properties of SIP request signals guarantee that any attempted request eventually will be granted. These properties depend only on the behaviour of the masters and the arbiter modules. The behaviour of the slave module does not affect the arbitration process. These liveness properties are shown in the following theorem:

Theorem 4 (Liveness of SIP: SIPLiveness_H)
SIP $\rightarrow$
$G((SnReset\ U\ SPnWait) \rightarrow F\ SPnWait)\ \wedge$
$G((\neg SInReadreq\ \wedge\ (SnReset\ U\ SInWait)) \rightarrow F\ SInWait)$

Theorem 4 describes two liveness properties of SIP. The first property deals with a request from the **Interrupt** module. When the module sends a request signal and no reset is applied during the wait process then the request will eventually

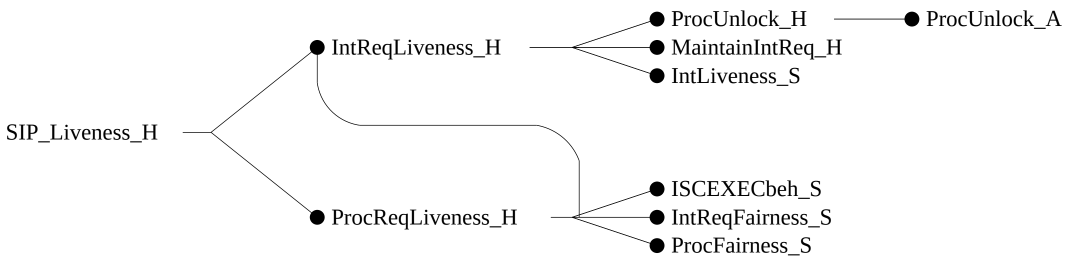

Fig. 7.1: Dependency lemmas for SIPLiveness

be granted. The second property states that the **processor** will eventually get ownership of the bus, provided no reset is applied during the wait process. The correctness of this theorem is given by two lemmas; the interrupt liveness lemma (Lemma 4) and the processor liveness lemma (Lemma 14). Figure 7.1 shows the lemmas needed to prove *SIPLiveness_H* (Theorem 4). A naming scheme is used for each lemma and theorems. A HOL lemma or theorem name is followed by _H. SMV and ACL2 lemmas or theorems are followed by _S and _A respectively.

All basic properties of the component are gathered using the appropriate formal tool. The choice of tool depends on how the component is specified. These properties are then gathered and imported into HOL, where the liveness of the SIP system will be proved.

In the remainder of this section, the proofs of these liveness lemmas are described. A sketch of proofs of the main liveness lemmas is provided. Figure 7.1 provides the road map for the structure of the descriptions.

7.1.1 Liveness of Interrupt Request

The first liveness property is the liveness of interrupt request. This property is described in Lemma 4. The lemma guarantees that any interrupt request will eventually be granted. The lemma is shown below:

Lemma 4 (Liveness of Interrupt Request: IntReqLiveness_H)
SIP $\rightarrow G((\neg SInReadreq \wedge (SnReset\ U\ SInWait)) \rightarrow F\ SInWait)$

The first lemma used to prove Lemma 4 is the processor unlock lemma. The SIP design states that a request from the **Interrupt** module has a higher priority than the **Processor** module bus request. Normally, the **Arbiter** grants the **interrupt** bus access in the following clock cycle after receiving the request signal. The only exception is when the **Processor** is locking the bus. The **Arbiter** has to wait until the **Processor** has completed its activity and retracts the lock signal before it can grant a new arbitration. The liveness condition requires the **Processor** to be able to retract its lock request and not to lock the bus forever. Lemma 5 shows that the processor will eventually retract the lock request. The lemma is shown below:

Lemma 5 (Processor not locking (ACL2): ProcUnlock_A)
$rst_1 \ \wedge \ wt_1 \ \wedge \ (natnp \ di_1) \ \wedge \ (statep \ s)$
$\rightarrow$
$\neg(PLock(PSCexec \ rst_1 \ wt_1 \ di_1 \ s)) \vee$
$(rst_2 \ \wedge \ wt_2 \ \wedge \ (natnp \ di_2)$
$\rightarrow$
$\neg PLock(PSCexec \ rst_2 \ wt_2 \ di_2 \ (PSCexec \ rst_1 \ wt_1 \ di_1 \ s)) \vee$
$(rst_3 \ \wedge \ wt_3 \ \wedge \ (natnp \ di_3)$
$\rightarrow$
$\neg PLock(PSCexec \ rst_3 \ wt_3 \ di_3 \ (PSCexec \ rst_2 \ wt_2 \ di_2 \ (PSCexec \ rst_1 \ wt_1 \ di_1 \ s)))$

The above lemma says that for any processor state, in at most three clock cycles the processor will un-lock the bus. The correctness of the above lemma is obtained by performing a case split analysis on all instruction cycles of the processor. The instruction cycles are described in Section 6.2. Based on the number of cycles to the un-locking condition, the case split results are grouped into three categories: one execution cycle; two execution cycles; and three execution cycles. Figure 7.2 shows the lemmas needed to prove *ProcUnlock_A* (Lemma 5).

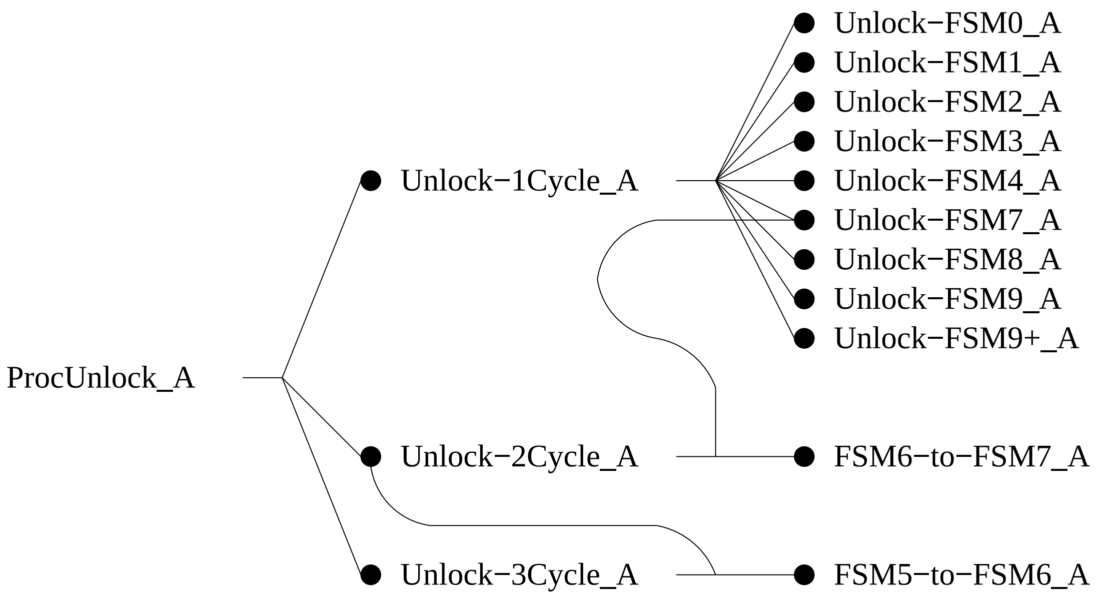

Fig. 7.2: Dependency lemmas for ProcUnlock_A

The Processor sends a lock (HIGH) signal only when it executes a SWAP instruction. It is indicated by an internal fsm set to 5 or 6. Otherwise, the lock signal is set to LOW which indicates that the processor is not in a lock mode. The later condition is described in Lemma 6.

Lemma 6 (Processor one cycle unlock: Unlock-1Cycle_A)
$rst \wedge wt \wedge (natnp\ di) \wedge (statep\ s) \wedge ((Pfsm\ s <= 4) \vee (Pfsm\ s >= 7))$
$\rightarrow$
$\neg PLock(PSCexec\ rst\ wt\ di\ s)$

Lemma 7 shows one of the cases used in proving Lemma 6. It states that with a no reset, a no wait, and an internal fsm equal to 7, a no lock signal will be produced as a result of the processor's execution.

Lemma 7 (Processor state 7: Unlock-FSM7_A)
$rst \wedge wt \wedge (natnp\ di) \wedge (statep\ s) \wedge (Pfsm\ s = 7) \rightarrow \neg PLock(PSCexec\ rst\ wt\ di\ s)$

There are eight more lemmas which are similar to Lemma 7. Each lemma represents a proof for each possible processor state, except when the state is equal to 5 or 6. These lemmas are used to prove *Unlock-1Cycle_A* (Lemma 6).

If the internal fsm of the processor module is equal to 6, then in the next cycle the internal fsm of the processor can be in state 7. This behaviour is described in Lemma 8.

Lemma 8 (Processor state 6: FSM6-to-FSM7_A)
$rst \wedge wt \wedge (natnp\ di) \wedge (statep\ s) \wedge (Pfsm\ s = 6) \rightarrow Pfsm(PSCexec\ rst\ wt\ di\ s) = 7$

Using a combination of Lemma 7 and 8, the *Unlock-2Cycle_A* lemma (Lemma 9) is proven. The lemma says that in two execution cycles, the processor will send a no lock signal to the arbiter.

Lemma 9 (Processor two cycle unlock: Unlock-2Cycle_A)
$rst_1 \wedge wt_1 \wedge (natnp\ di_1) \wedge rst_2 \wedge wt_2 \wedge (natnp\ di_2) \wedge (statep\ s) \wedge (Pfsm\ s = 6)$
$\rightarrow$
$\neg PLock(PSCexec\ rst_2\ wt_2\ di_2\ (PSCexec\ rst_1\ wt_1\ di_1\ s))$

A similar proof will be needed for the processor execution that starts in state 5. In this case the first evaluation changes the processor's state into state 6. This condition results in a need for a minimum of three processor cycles to reach the un-lock condition. This behaviour is shown in Lemma 10.

Lemma 10 (Processor three cycle unlock: Unlock-3Cycle_A)
$rst_1 \wedge wt_1 \wedge (natnp\ di_1) \wedge rst_2 \wedge wt_2 \wedge (natnp\ di_2) \wedge$
$rst_3 \wedge wt_3 \wedge (natnp\ di_3) \wedge (statep\ s) \wedge (Pfsm\ s = 5)$
$\rightarrow$
$\neg PLock(PSCexec\ rst_3\ wt_3\ di_3\ (PSCexec\ rst_2\ wt_2\ di_2\ (PSCexec\ rst_1\ wt_1\ di_1\ s)))$

The *ProcUnlock_A* lemma (Lemma 5) and its subsequent lemmas are proved using the ACL2 theorem prover. Lemma 5 is imported into HOL and transformed using HOL's **Processor** module definition. The HOL code to import and transform Lemma 5 is shown as follows:

```
val ProcUnlock_A = GEN_ALL (getthm [] "PROCUNLOCK");
val ProcUnlock_H = TAC_PROOF(([],--`
  !SnReset SPnWait SPDi PS0 SPState SPnMreq SPLock SPRW SPDo SPAdd.
  Processor SnReset SPnWait SPDi PS0 SPState SPnMreq SPLock SPRW SPDo SPAdd
  ==>
  ALWAYS
   (\t. ((SnReset t) /\ (SPnWait t)) ==>
        (~(SPLock t) \/
         ((NEXT (\tx. (SnReset tx) /\ (SPnWait tx)) t) ==>
          (~(NEXT SPLock t)) \/
           ((NEXT (NEXT (\tx. (SnReset tx) /\ (SPnWait tx))) t) ==>
            ~(NEXT (NEXT SPLock) t))))) 0`--),
  ...
  (ASSUME_TAC SPCGivesStatep_H) THEN ...
  (ASSUME_TAC (SPECL ... ProcUnlock_A)) THEN ... );
```

The *PROCUNLOCK* lemma which is proven in ACL2 is imported into HOL and stored as *ProcUnlock_A*. Lemma *ProcUnlock_A* and Lemma 3 are used to prove *ProcUnlock_H* lemma. The *ProcUnlock_H* lemma is described in Lemma 11.

Lemma 11 (Processor not locking (HOL): ProcUnlock_H)
Processor

$$\begin{array}{l}
\rightarrow \\
G(SnReset \;\wedge\; SPnWait \\
\quad \rightarrow \\
\quad \neg SPLock \;\vee\; (X(SnReset \;\wedge\; SPnWait) \\
\qquad\qquad\qquad \rightarrow \\
\qquad\qquad\qquad X\neg SPLock \;\vee\; (X^2(SnReset \;\wedge\; SPnWait) \\
\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad X^2(\neg SPLock))))
\end{array}$$

The lemma guarantees that in at most three continous processor execution cycles, the processor will unlock the bus.

The second lemma used to prove Lemma 4 is the *continuation of interrupt request* lemma. When the system is in lock mode, the **Interrupt** is forced to be in a wait state. The request from the interrupt will be evaluated again in the next clock cycle. Since **Arbiter** does not memorise or keep track of incoming requests, the **Interrupt** has to send the request signal again in the next clock cycle. This interrupt property is shown in the following lemma:

Lemma 12 (Interrupt maintains its request: MaintainIntReq_H)
Interrupt $\rightarrow G((X\ SnReset \;\wedge\; \neg SInReadreq \;\wedge\; X\neg SInWait) \rightarrow X\neg SInReadreq)$

The third lemma used to prove Lemma 4 is the *liveness of interrupt request* lemma. A combination of **Arbiter** and properties from Lemma 11 and 12 are used to show the requirements to grant an interrupt request. When the **interrupt** maintains its request until it is granted and the **processor** is not infinetly locking the bus then the **Arbiter** will eventually grant the interrupt request. This condition is described in Lemma 13. We prove this lemma using SMV. The result is as follows:

Lemma 13 (Liveness condition of Interrupt Request: IntLiveness_S)
Arbiter $\wedge$
$G((X\ SnReset\ \wedge\ \neg SInReadreq\ \wedge\ X\neg SInWait) \rightarrow X\neg SInReadreq)\ \wedge$
$G(SnReset\ \wedge\ SPnWait$
$\quad\rightarrow$
$\quad\neg SPLock\ \vee\ (X(SnReset\ \wedge\ SPnWait)$
$\qquad\qquad\rightarrow$
$\qquad\qquad X\neg SPLock\ \vee\ (X^2(SnReset\ \wedge\ SPnWait)$
$\qquad\qquad\qquad\qquad\rightarrow$
$\qquad\qquad\qquad\qquad X^2(\neg SPLock))))$
$\rightarrow$
$G((\neg SInReadreq\ \wedge\ (SnReset\ U\ SInWait)) \rightarrow F\ SInWait)$

By combining Lemma 11, Lemma 12, and Lemma 13, the system level liveness of the interrupt module described in Lemma 4 can be proven. The lemma is interpreted as follows: If we have a system which contains a **Processor**, an **Interrupt**, and an **Arbiter**, then whenever the interrupt sends a request signal it will eventually be granted.

7.1.2 Liveness of Processor Request

The second liveness property is the liveness of the processor module to access the bus. The lemma is shown below:

Lemma 14 (Liveness of Processor Request: ProcReqLiveness_H)
SIP $\rightarrow G((SnReset\ U\ SPnWait)) \rightarrow F\ SPnWait)$

There are three conditions for the arbiter to grant the processor module bus access. First, the processor sends a lock request signal to the arbiter which recieves the highest arbitration priority. Second, when the processor sends a request signal and the interrupt is in an idle condition then the processor is granted bus access. Third, when no master is requesting to access the bus, the arbiter grants the default master/processor. In other words, the only condition that prevents the processor from accessing the bus is when the processor is not locking the bus and the interrupt is requesting to access the bus. These conditions are used to show the liveness of the processor module. Lemma 15 shows that the interrupt will eventually stop requesting for bus access. This property is obtained by model checking the behaviour of the interrupt module.

Lemma 15 (Interrupt behaviour: ISCEXECbeh_S)
Interrupt $\rightarrow G(GF\ SInWait \rightarrow F\ SInReadreq)$

In this stage, two properties of the interrupt module have been proven. First, when the interrupt is frequently granted then it will eventually stop requesting for access to the bus (Lemma 15). Second, when the interrupt is requesting the bus and no reset is applied during the waiting period then the request will eventually be granted (Lemma 4). A new property about the interrupt is derived: when there is no reset then the interrupt will eventually stop requesting for access to the bus. This property is shown in Lemma 16. SMV is used to prove the lemma.

Lemma 16 (Fairness of interrupt request: IntReqFairness_S)
$G(GF\ SInWait \rightarrow F\ SInReadreq) \wedge$
$G((\neg SInReadreq\ \wedge\ (SnReset\ U\ SInWait)) \rightarrow F\ SInWait)$
$\rightarrow$
$G((SnReset\ U\ SInReadreq) \rightarrow F\ SInReadreq)$

The *Fairness of interrupt request* lemma is used to prove that eventually the processor is granted bus access. This statement is proved using an SMV model checker. The verification result is described in Lemma 17.

Lemma 17 (Processor fair process: ProcFairness_S)
Arbiter $\wedge$
$G((SnReset\ U\ SInReadreq) \rightarrow F\ SInReadreq)$
$\rightarrow$
$G((SnReset\ U\ SPnWait) \rightarrow F\ SPnWait)$

Combining Lemma 4, 15, 16, and 17 proves the system level liveness of the processor module described in Lemma 14. If a system contains a **Processor**, an **Interrupt**, and an **Arbiter**, then the processor module will eventually get the ownership of the bus.

7.2 Hardware/Software Correctness

In a full-chip validation, the correctness of the SIP running an application software is verified. A simple arithmetic program as described in Section 6.4.3

is stored in the memory module. The full-chip validation targets the correctness verification of the program running in the SIP. In the verification process, two basic assumptions are made. First, the code is a non self-modifying program. One possible scenario is to store the program in a ROM device. Second, all data needed during the process is already available in the memory. No analysis on the off-chip process will be performed in this verification.

Two enviromental constraints are needed to show that the application will work correctly in the system. First, the system is assumed to be about to execute the application. Second, the memory is partitioned into two groups: the first group is used to store the application code and the data used in the execution of the program; the second group is for the remainder of the memory system. The memory module has been designed to support the requirements of two memory groups. The first group of memory $SMState_1$ is defined to be used by the application. The others are stored in the second group of memory $SMState_2$. Four enviromental assumptions are defined:

- No reset is applied during the analysis.
- When the Interrupt is active, no write process is allowed in $SMState_1$.
- The processor has to be able to execute the application code.
- All external data needed by the processor are ready in $SMState_1$.

Using these four assumptions, the correctness of the application running in the system can be proved. The correctness proof of the application running on the SIP is described in Theorem 5.

Theorem 5 (Full-Chip validation: SIPSWHWSim_H)
SIP $\wedge$
$(\mathit{NATNP}\ X_1) \wedge$
$(\mathit{NATNP}\ X_2) \wedge$
$\forall t.\ (\mathit{SnReset}\ t) \wedge$
$\forall t.\ (\mathit{SPnWait}\ t) \rightarrow (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMState}_1\ t))) \wedge$
$\forall t.\ (\mathit{SInWait}\ t) \rightarrow (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMState}_1\ t)) \rightarrow \neg(\mathit{SInRW}\ t))$
$\rightarrow$
$\forall t.\ (\mathit{INITSTATE}(\mathit{SPState}\ t)) \wedge$
$(\mathit{MEMLIST}(\mathit{SMState}_1\ t) = \mathit{MEMRESET}\ X_1\ X_2)$
$\rightarrow$
$\exists t_x.\ \mathit{SOFTWAREOK}\ X_1\ X_2\ (\mathit{MEMLIST}(\mathit{SMState}_1(t + t_x)))$

The verification is performed by splitting the process into two stages. In the first stage, the system is partitioned by selecting the software component and its execution engine, with the verification performed for the selected components. In the second stage, facts of the partitioned system are mapped back to the complete system.

The *Full-Chip validation* uses two main lemmas, the *processor-memory simulation* lemma (Lemma 18) and the *SIP multicycle simulation* lemma (Lemma 19). The first lemma shows the correctness of application executed by only the processor and the memory modules. The second lemma shows the fairness of any instruction executed in the system. Figure 7.3 shows the lemmas needed to prove the *full-chip validation* theorem (Theorem 5)

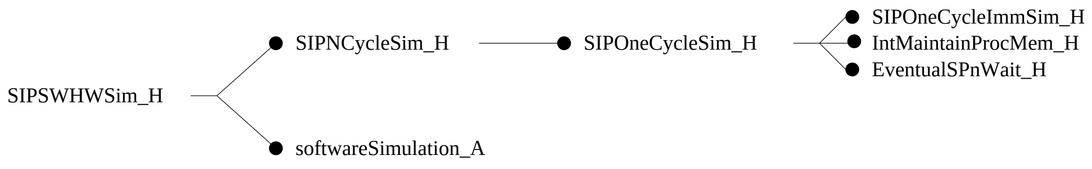

Fig. 7.3: Dependency lemmas for SIPSWHWSim_H

7.2.1 Processor-Memory Simulation

Both the processor and memory modules are modelled in ACL2. These two modules are combined using an intermediate function *OneCycleSim* (Definition 34). This function provides the connectivity and operational sequences between inputs and outputs to and from these two modules. The sequential process of *OneCycleSim* may be described in two half-cycles. In the first half-cycle, the processor executes. If the processor is performing a write operation, both the address and data-output are available otherwise only the address is valid. The second step is memory execution. When the memory performs a write operation, it takes the address and data-output from the processor, otherwise it uses the address to obtain the contents of the memory indicated by the address. In the second half-cyle, the processor uses the data-output from the memory to complete its evaluation process.

The result of evaluating *OneCycleSim* is an updated state of the processor and memory. A new datatype *XVPState* is defined to represent the processor state and the memory state as a tuple. The first element of the tuple is the state of the processor and the second element is the state of the memory.

Definition 34 (ACL2 P-M single cycle evaluation: OneCycleSim)

```
$OneCycleSim\ rst\ wt\ simst$
$\overset{def}{=}$
$let\ TempDo\ (Pdo(PSCexec_l\ rst\ wt\ (xpm\ simst)))$
    $TempAdd\ (Padd(PSCexec_l\ rst\ wt\ (xpm\ simst)))$
    $TempRW\ (Prw(PSCexec_l\ rst\ wt\ (xpm\ simst)))$
    $TempPState\ (PSCexec_l\ rst\ wt\ (xpm\ simst))$
    $NextDo\ (MemExecR\ rst\ (Padd(PSCexec_l\ rst\ wt\ (xpm\ simst)))(xmmsimst))$
$if\ wt$
$then\ if\ TempRW$
      $then\ (XVPState\ (PSCexec_h\ rst\ wt\ TempDo\ TempPState)$
                      $(MemExecW\ rst\ TempDo\ TempAdd\ memst))$
      $else\ (XVPState\ (PSCexec_h\ rst\ wt\ NextDo\ TempPState)(xmm\ simst)$
$else\ simst$
```

In simulation, the termination condition is determined by defining how many cycles take place in the evaluation. This termination condition is defined in Definition 35. During this evaluation, it is assumed that the system executes continously without any interruption until the termination condition is achieved. Because both reset and wait signals are active low, they are set to HIGH or TRUE.

Definition 35 (ACL2 P-M N cycle simulation: NCycleSim)

```
$NCycleSim\ N\ XVPstate$
$\overset{def}{=}$
$if\ (N = 0)$
$then\ XVPstate$
$else\ NCycleSim\ (N - 1)\ (OneCycleSim\ T\ T\ XVPstate)$
```

In this stage, system precondition requirements are defined so that the simulation or evaluation of the targeted code is about to start. The requirements define the initial state of the processor module and the memory module. The requirements are defined as follows:

- The internal state machine indicates that in the next cycle the next instruction in the pipeline will be executed.
- The processor is not in reset mode.
- The processor is not in flushing pipeline mode.
- The top instruction in the pipeline is the first code to be evaluated.
- The second instruction in the queue is the second code to be executed.

- The address-bus indicates the address of the second code in the external memory.

The processor's initial condition is defined in Definition 36.

Definition 36 (ACL2 State of processor constraints: initPState)

$$\begin{aligned}
&\mathit{initState\ Pstate}\\
&\overset{def}{=}\\
&(\mathit{Pp1\ s} = (\mathit{MOV\ R_5, \#10})) \land\\
&(\mathit{Pp2\ s} = (\mathit{LDA\ R_0, R_5})) \land\\
&(\mathit{Ppf\ s} = \mathit{0}) \land\\
&(\mathit{Pfsm\ s} = \mathit{1}) \land\\
&(\mathit{Padd\ s} = \mathit{2}) \land\\
&\neg(\mathit{Preset\ s})
\end{aligned}$$

The initial state of the memory module (initMstate X_1X_2) is interpreted as the default condition. It contains only the code or program and all data to be processed. The function *initMstate* adds symbolic data X_1 and X_2 to the default memory state.

Finally, the correctness criteria of the code is defined. The code which is defined in Section 6.4.3 takes two values (X_1, X_2), adds them together, and writes back to the memory. Function *softwareOK* contains the procedure to check the correctness of the final contents of the memory.

Definition 37 (ACL2 Simulation correctness criteria: softwareOK)

$$\begin{aligned}
&\mathit{softwareOK\ X_1\ X_2\ Mstate}\\
&\overset{def}{=}\\
&(\mathit{getMemMemst\ 16\ Mstate}) = \mathit{X_1} \land\\
&(\mathit{getMemMemst\ 17\ Mstate}) = \mathit{X_2} \land\\
&(\mathit{getMemMemst\ 18\ Mstate}) = (\mathit{X_1} + \mathit{X_2})
\end{aligned}$$

Lemma 18 shows that when the initial conditions of the processor and memory are satisfied, the evaluation of the code executed by the processor satisfies the correctness criteria. The lemma was proven in ACL2.

Lemma 18 (Processor-Memory simulation: softwareSimulation_A)

$$\begin{aligned}
&(\mathit{natnp\ X_1}) \land (\mathit{natnp\ X_2}) \land (\mathit{statep\ Pstate}) \land (\mathit{initState\ s})\\
&\rightarrow\\
&\mathit{softwareOK\ X_1\ X_2\ XMM}(\mathit{NCycleSim\ 15\ }(\mathit{XVPState\ s\ }(\mathit{initMstate\ X_1\ X_2})))
\end{aligned}$$

7.2.2 SIP Simulation

The single-cycle processor-memory evaluation depends on the environmental assumption that the processor is granted access to the bus. Similarly, the multi-cycle evaluation uses the same assumptions that the processor is always granted to continue its execution. In SIP, this assumption is not always true. There is a possibility that the processor has to wait for a few cycles before it becomes active. The *processor liveness* lemma (Lemma 14) guarantees that eventually the processor will be active. Suppose at time = t, the processor is ready to evaluate a valid state of processor *SPState* and memory $SMState_1$ then the valid result will eventually appear after t_x cycle. This property is described in the following lemma:

Lemma 19 (SIP multicycle simulation: SIPNCycleSim_H)
SIP $\wedge$
$(NATNP\ N) \wedge$
$\forall t.\ (SnReset\ t) \wedge$
$\forall t.\ (SPnWait\ t) \rightarrow (ISMEMEXISTS(SMAdd\ t)(MEMLIST(SMState_1\ t))) \wedge$
$\forall t.\ (SInWait\ t) \rightarrow (ISMEMEXISTS(SMAdd\ t)(MEMLIST(SMState_1\ t)) \rightarrow \neg(SInRW\ t))$
$\rightarrow$
$\forall t. \exists t_x.$
$EXECUNTIL\ N\ (XVPSTATE\ (SPState\ t)(MEMLIST(SMState_1\ t)))$
$=$
$XVPSTATE(SPState(t + t_x))(MEMLIST(SMState_1(t + t_x)))$

The *multicycle evaluation* lemma (Lemma 19) is an extension of the *single cycle evaluation* lemma (Lemma 20). The interpretation of Lemma 20 is similar to Lemma 19, the only difference is in the number of cycles. Lemma 20 is the lemma for one processor execution cycle while Lemma 19 is the lemma for N processor execution cycles. The *single cycle evaluation* lemma is shown below:

Lemma 20 (SIP one cycle simulation: SIPOneCycleSim_H)
SIP $\wedge$
$\forall t.\ (SnReset\ t) \wedge$
$\forall t.\ (SPnWait\ t) \rightarrow (ISMEMEXISTS(SMAdd\ t)(MEMLIST(SMState_1\ t))) \wedge$
$\forall t.\ (SInWait\ t) \rightarrow (ISMEMEXISTS(SMAdd\ t)(MEMLIST(SMState_1\ t)) \rightarrow \neg(SInRW\ t))$
$\rightarrow$
$\forall t. \exists t_x.$
$ONECYCLESIM\ \ T\ \ T\ (XVPSTATE(SPState\ t)(MEMLIST(SMState_1\ t)))$
$=$
$XVPSTATE(SPState(t + t_x))(MEMLIST(SMState_1(t + t_x)))$

The next three lemmas are used to prove Lemma 20. These lemmas describe the condition when the processor is granted the bus and when it is set to idle.

First, in ACL2 we assume that at time = t the processor is granted access to the bus and no reset is applied, then by definition the result of this evaluation will appear in the next cycle (t+1). The property is defined in HOL and described in Lemma 21. The description of the lemma is as follows: at time t the processor is not in the wait mode and no reset is applied, then the result of evaluating the SIP will appear at (t+1). This property is shown in Lemma 21.

Lemma 21 (SIP one cycle immediate simulation: SIPOneCycleImmSim_H)

$$
\begin{array}{l}
\textbf{SIP} \wedge \\
\forall t.\ (\mathit{SnReset}\ t) \wedge \\
\forall t.\ (\mathit{SPnWait}\ t) \rightarrow (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMState_1}\ t))) \\
\rightarrow \\
\forall t.\ (\mathit{SPnWait}\ t) \\
\quad \rightarrow \\
\quad \mathit{ONECYCLESIM}\ T\ T\ (\mathit{XVPSTATE}(\mathit{SPState}\ t)(\mathit{MEMLIST}(\mathit{SMState_1}\ t)) \\
\quad = \\
\quad \mathit{XVPSTATE}(\mathit{SPState}(t+1))(\mathit{MEMLIST}(\mathit{SMState_1}(t+1)))
\end{array}
$$

Second, consider the possibility that the processor is in the wait state. In the wait/idle state, the processor maintains its internal state. The property is proved in ACL2. Memory $\mathit{SMState}_1$ is used primarily by the processor. The interrupt can only read the data without the possibility to write it. In this case, the memory state of $\mathit{SMState}_1$ is not changed. These two properties are gathered in HOL. They are used to show that when the arbiter did not grant the processor, the processor's state (*SPState*) and the memory's state ($\mathit{SMState}_1$) remain the same. These properties are described in Lemma 22.

Lemma 22 (Processor Memory maintain states: IntMaintainProcMem_H)

$$
\begin{array}{l}
\textbf{Arbiter} \wedge \\
\forall t.\ (\mathit{SnReset}\ t) \\
\forall t.\ \neg(\mathit{SPnWait}\ t) \\
\quad \rightarrow \\
\quad (\mathit{ISMEMEXISTS}(\mathit{SMAdd}\ t)(\mathit{MEMLIST}(\mathit{SMState_1}\ t)) \rightarrow \neg(\mathit{SInRW}\ t)) \\
\rightarrow \\
\forall t.\ \neg(\mathit{SPnWait}\ t) \\
\quad \rightarrow \\
\quad \mathit{XVPSTATE}(\mathit{SPState}\ t)(\mathit{MEMLIST}(\mathit{SMState_1}\ t)) \\
\quad = \\
\quad \mathit{XVPSTATE}(\mathit{SPState}(t+1))(\mathit{MEMLIST}(\mathit{SMState_1}(t+1)))
\end{array}
$$

Finally, the interrupt module has to have a fairness behaviour by not infinitely requesting the bus. The processor liveness lemma (Lemma 17) guarantees that the processor eventually gets the bus. A more detailed analysis of the behaviour of the interrupt module shows that the processor will get the bus in four cycles at the most. This final property is shown below:

Lemma 23 (Fairness of Idle Cycle: EventualSPnWait_H)
SIP $\wedge$
$\forall t.\ (\mathit{SnReset}\ t)$
$\rightarrow$
$\forall t.\ (\mathit{SPnWait}(t+\mathit{1}))\ \vee\ (\mathit{SPnWait}(t+\mathit{2}))\ \vee\ (\mathit{SPnWait}(t+\mathit{3}))\ \vee\ (\mathit{SPnWait}(t+\mathit{4})$

Using Lemma 21, 22 and 23, the *one cycle simulation* lemma (Lemma 20) is proven. This lemma is the bases of the *multi cycle simulation* lemma (Lemma 19). The generic form of Lemma 19 provides a guarantee that any program correctly executed using the processor will always preserve its correctness in the platform.

7.3 Summary

The case study presented in this chapter is aimed at addressing one problem in the verification of SoC design; given a complete SoC system, how to validate the system as a whole. In this chapter, system level verifications of the SIP were demonstrated. It shows that it is possible to perform system level reasoning to a complex embedded system which includes software execution. Within an integrated heterogenous formal verification framework, combining results from several tools can be done effectively.

Two system level properties to be validated were defined: the liveness of every master request; and the software correctness when the software is embedded in the system. The liveness verification is achieved by analysing each master request. The validation starts from the highest priority module to the lowest. It was shown that each request can eventually be granted and every process can be completed to allow a new arbitration. The software correctness verification was achieved by decomposing the problem into two subgoals. First, the correctness of the software component with its execution engine is validated. Second, the platform will always correctly execute any processor instruction. The combination of these two properties shows the correctness of the software component executed using the platform.

The SIP was verified on a Linux machine with an Intel Pentium II 400MHz processor with 384M RAM. The processor model was built in one minute. Twenty one lemmas were used to verify the processor unlock lemma. The verification was completed in one minute. The SIP liveness properties were verified in two minutes and used fourteen lemmas. The proof distribution of these lemmas is as follows: two lemmas are proved in ACL2; four lemmas are proved in SMV; and eight are proved in HOL. The hardware/software verification used fifty five lemmas. The proof distribution of these lemmas is as follows: twenty lemmas are proved in ACL2; six lemmas are proved in SMV; and twenty nine lemmas are proved in HOL. The verification was completed in two and a half minutes.

Chapter 8

The ARM Integration and Verification Platform

The integration platform design methodology allows designers to mix and match reusable virtual components. A similar idea is adopted for the formal verification platform methodology. In this methodology, formal systems are developed by integrating reusable formal components. The properties of the system are then verified using the formal verification environment. A methodology is defined to develop a generic and reusable formal verification platform. The generality of the platform makes it reusable in the development of an application specific platform.

In this chapter, the basic components used in the development of an ARM formal verification platform will be described. The platform is based on an ARM AMBA bus protocol and an ARM7 processor. In Section 8.1, a description of the ARM AMBA bus protocol will be given. Specifically, the RAPIER's AMBA AHB bus protocol will be discussed. In Section 8.2, a description of the ARM7 processor model and its wrapper will be provided.

8.1 ARM AMBA

8.1.1 AMBA Specification

The *Advanced Micro-controller Bus Architecture* (AMBA) is an on-chip bus specification that defines the interconnection, communication, and management of functional blocks for SoC design. The AMBA specification facilitates the *right-first-time* development of SoC design. It is based on technology independent specification. This ensures that the modules are highly reusable across diverse IC processes and technologies. It encourages standardised modular system design using common bus protocols, enhancing the reuse design methodology for the modules.

There are three types of AMBA busses:

- The *Advanced High-Performance Bus* (AHB)
- The *Advanced System Bus* (ASB)
- The *Advanced Peripheral Bus* (APB)

The AMBA AHB is used for the design of high performance and high frequency system modules. The AMBA ASB is used only for designing high performance system modules. Both types of busses are used to support efficient connection of processors, on-chip memories, and off-chip external memory interfaces. The AMBA ASB supports only pipeline operations and multiple bus masters. In addition to AMBA ASB features, AMBA AHB also supports burst transfers and split transactions. A burst transfer is one or more data transactions requested by a bus master. A split transfer is an incomplete transfer which requires a bus master to retry the transfer. A slave requests access to the bus on behalf of the master when it decides that the split transfer can be completed. The AMBA APB is used for the design of low-power peripherals and is used in conjunction with either AMBA AHB or AMBA ASB.

Typically, an AMBA based SoC design contains a high performance bus system such as an AMBA AHB or an AMBA ASB. The bus is capable of handling a high-bandwidth process with an external memory interface, processor, on-chip memory, *Direct Memory Access* (DMA) module, and a bridge to a lower speed bus APB where most peripherals in the system are located. The block diagram of a typical AMBA system is shown in Figure 8.1. The block diagram is very similar to the one described in Figure 4.1. In this figure, the AMBA AHB or AMBA ASB and AMBA APB are represented by the PLB and OPB.

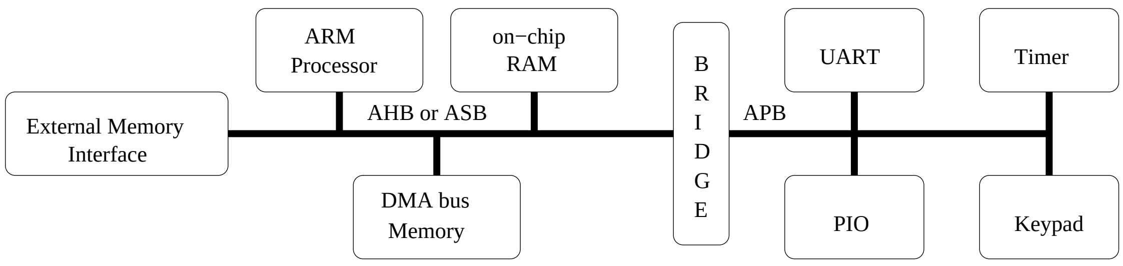

Fig. 8.1: Typical AMBA system

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 of AHB slaves. Most of the peripherals can be part of the system of AHB slaves. More likely, they are part of the AMBA APB.

In this case study, the AMBA AHB was used to demonstrate how to generate reusable formal models and formal proofs. The reuse of formal proofs are shown in defining a specific application. For the rest of the section, the discussion will focus on the AMBA AHB.

8.1.2 AMBA AHB

The AMBA AHB is a high performance bus system which provides high bandwidth operations and supports multiple bus masters. A typical AMBA AHB system contains the following four components: AHB master, AHB slave, AHB arbiter, and AHB decoder. An AHB master is a bus master which provides the necessary control signals and data to trigger read or write operations. An AHB slave is a bus slave which responds to the operation initiated by the bus master. It replies back to the active master on its operational status, either successes or failures. An AHB arbiter is the bus arbiter which controls the arbitration of master processes. In AMBA, the bus arbiter ensures that at any time there will be one and only one bus master which can start read or write operations. An AHB decoder is the module that decodes the address for each transfer and provides selection signals to indicate which signals are going through.

Interfaces

All control and response signals from the AHB masters and AHB slaves are managed by the AHB arbiter. In total the AHB arbiter has 18 signal interfaces, of which nine signals originate from the masters and four signals from the slaves. There are three signals coming out from the arbiter, one control signal from the decoder, and one system reset signal. These signals are described in Table 8.1.

Bus Architecture

The bus architecture in the AMBA AHB uses a central multiplexer interconnection scheme. There are three bus multiplexers, one each for: write data (HWdata), read data (HRdata), and address (HAddr). The AHB arbiter is responsible for deciding which master's data (HWdata or HRdata) is going through the multiplexer to the slaves.

Name	Source	Description
HReset	system	is the reset signal. A LOW level signal forces the system to go to the default reset state
HAddr$_x$	master	is the bus address
HTrans$_x$	master	is the type of data transfer It can be either *nonseq*, *seq*, *idle*, or *busy*
HWrite$_x$	master	is the read or write data transfer control signal. Read operation sets the signal to LOW, and vice-versa.
HSize$_x$	master	is the size of data.
HBurst$_x$	master	is the number of data transfer.
HProt$_x$	master	is the protection control signals for master module
HWdata$_x$	master	The data bus for transfer data from master to slave during the write operation
HSel$_x$	decoder	is the select signal to indicate the active slave
HRdata	slave	is the data bus for transfer data from slave to master during the read operation
HReady	slave	is the status of data transfer signal. A HIGH signal indicates a transfer is completed and vice-versa
HResp	slave	is the status responses signal for data transfer. It can be either *okay*, *error*, *retry*, or *split*
HBusreq$_x$	master	The request signal from master module
HLock$_x$	master	The lock request signal from master module
HGrant$_x$	arbiter	The grant signal for master x that it gets the bus access
HMaster	arbiter	is the current active master indicators
HMasterlock	arbiter	is the arbiter's lock indicator signal
HSplit$_x$	slave	is the slave signal to indicate which master is allowed to resume the split transaction

Tab. 8.1: AMBA AHB Interfaces

Overview of AMBA AHB operation

The AMBA AHB process starts when the AHB masters assert bus request signals (HBusreq$_x$) to the AHB arbiters. The AHB arbiters then perform the arbitration process to determine which master is granted (HGrant$_x$) access to the bus. The selected master starts the data transfer process by sending the control signal and address (HAddr). The control signals provide information for the transfer. They are grouped in the following categories:

- Type of transfer (HTrans). There are four types of transfer: *nonseq*, *seq*, *idle*, and *busy*. The *nonseq* signal indicates that the first transfer of a burst is not related to the previous transfer. The *seq* signal indicates that the remaining burst transfers are sequential. The control signal and address relate to the previous one. In an *idle* transfer, the active master does not perform any data transfer. The *busy* transfer is similar to the *idle* cycle. It indicates that the master is inserting an *idle* cycle in the middle of the burst operation. For *idle* and *busy* transfers, the slave must respond with the *okay* signal.

- The burst operation (HBurst). There are eight types of burst operation: *single*, *incr*, *wrap4*, *incr4*, *wrap8*, *incr8*, *wrap16*, and *incr16*. In a *wrap* burst, the address is wrapped when the boundary is reached. For example, a *wrap4* burst transfer will wrap at 16 byte boundaries. If the transfer started at the address 0x44, then the transfer will continue with the address 0x48, 0x4C, and 0x40.

- The data size (HSize). There are eight data sizes for the transfer. The sizes are 8, 16, 32, 64, 128, 256, 512, and 1024 bits. The sizes are used in conjunction with the burst signals to determine the address boundary for wrapping the burst operation.

- The protection control (HProt). There are eight types of protection signal such as the opcode fetch, data access, user access, privilege access, not bufferable, bufferable, not cache-able, cache-able. This protection signal is only used when some level of protection will be implemented. In most cases, this signal is left unused.

- The transfer direction (HWrite). This signal indicates the type of operation that the master is performing. It can be either a *read* or a *write* operation.

When the active master has started the transfer, the active slave responds with information on the transfer using HReady and HResp signals. Whenever the active slave needs to assert one or more wait states, the HReady signal is set to LOW. The HResp signal is used to determine the status of the transfer. There are four possible HResp responses: *nonseq*, *seq*, *idle*, and *busy*. The *okay* response on HResp indicates that the slave's transfer is progressing without any problems. When the transfer is completed the HReady is set to HIGH. The *error* response indicates that an error has occurred during the transfer. The *error* message is sent to the bus master so that it knows that the transfer is not successful. 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 attempt to retry the transfer when it is next granted access to the bus. In a *split* condition, the slave will request the access of the bus on the behalf of the master when the transfer can be completed. The *retry* and *split* responses allow slaves to delay the completion of the transfer along with making the bus free to be used by other masters.

The arbiter manages the arbitration processes. It monitors requests to access the bus from masters and complete the split transfer from slaves. Then it decides which master has the highest priority to be granted access. The arbitration process can be implemented using the highest priority, fair access, or a combination of both schemes. The arbiter is also responsible for guaranteeing that at any time there is only one master granted access to the bus.

8.1.3 Formal Model of AMBA AHB

The AMBA bus protocol described in this case study is based on the RAPIER AMBA implementation from ISLI. The bus protocol is a part of the ISLI foundation block system [53]. It is developed to provide an infrastructure for academic research projects and teaching platforms. The block diagram of the ISLI foundation block is shown in Figure 8.2.

The foundation block contains a two-bus architecture based on the AMBA AHB and the AMBA APB. There are five AMBA AHB ports which can be used by master modules such as the processor, *Digital Signal Processing* (DSP) processor, *Direct Memory Access* (DMA) controller, etc. The arbiter is responsible for determining which master is currently active. Slave modules are connected either to the AHB bus or the APB bus. High performance slave modules, such as on-chip RAM (Static RAM) and external memory controller are connected to the AHB bus. The APB is used by slower peripherals such as Timers, *Universal Asynchronous Receiver/Transmitter* (UART), *General Purpose Input/Output* (GPIO), and Interrupt and System controllers. The AHB bus and APB bus are connected through the AHB-APB bridge.

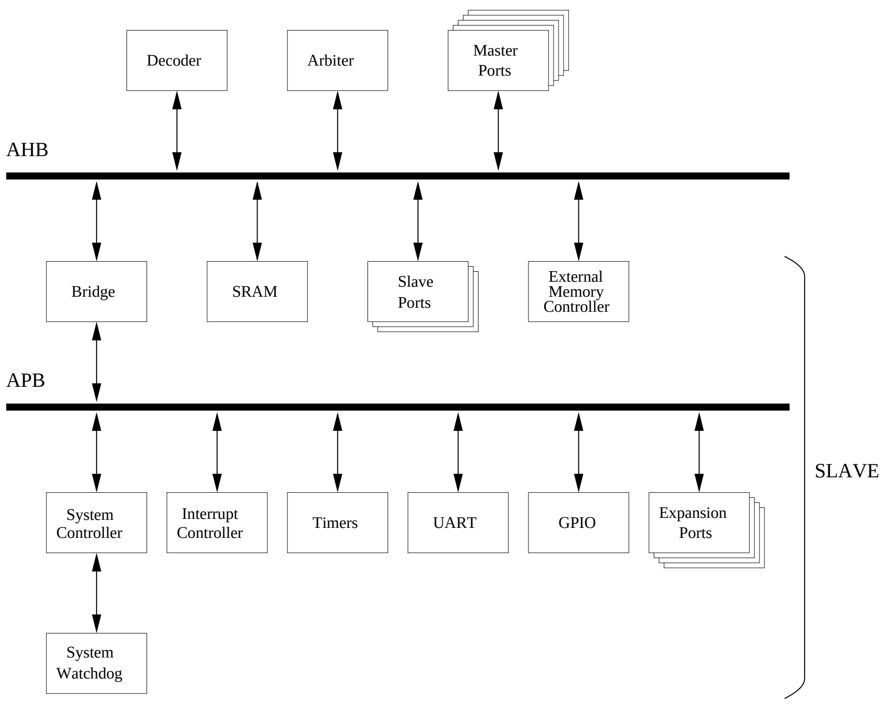

Fig. 8.2: ISLI Foundation Block System

Similar to the foundation block, the formal verification platform is used to integrate formal descriptions of the components and to perform system level formal verification. The platform is built based on the RAPIER foundation block. The platform centres on the AHB bus. It contains the five master ports, the bus controller, and a slave port. One slave port is introduced by generalising slave behaviour which, in fact, is the same.

The ISLI's AMBA AHB bus protocol is implemented in Verilog. The SMV model checker is the most suitable tool to do a formal check of this protocol. The Verilog code of ISLI's AMBA AHB must be transformed to the specification language of SMV, the SMV Language (SMVL), before it is model checked. The translation from Verilog to SMVL is done by using the translation tool *vl2smv* [56]. The inclusion of data and address paths in the model may cause a state space explosion which will prevent the model checker from completing the verification. One way to reduce the state space explosion is by abstracting these paths. In SMV, data abstraction can be achieved by declaring the n-bit bus signal as a scalar-set data-type called num [44].

AMBA-AHB interfaces contain fifty five input signals and twelve output signals. The input signals are hreset_i, hwrite_{ix}, htrans_{ix}, hsize_{ix}, hburst_{ix}, hprot_{ix}, hbusreq_{ix}, hlock_{ix}, haddr_{ix}, hwdata_{ix}, clken_{ix}, hready_i, hresp_i, hrdata_i, and hsplit$_i$. The output signals are hgrant$_{ox}$, hrdata$_o$, hmaster$_o$, hmastlock$_o$, hwrite$_o$, haddr$_o$, and hwdata$_o$. The interface representation of the AHB module is as follows:

> **AHB**$(hreset_i,\ hwrite_{i1},\ htrans_{i1},\ hsize_{i1},\ hburst_{i1},\ hprot_{i1},\ hbusreq_{i1},\ hlock_{i1},\ haddr_{i1},\ hwdata_{i1},\ hgrant_{o1},\ clken_{i1},\ hwrite_{i2},\ htrans_{i2},\ hsize_{i2},\ hburst_{i2},\ hprot_{i2},\ hbusreq_{i2},\ hlock_{i2},\ haddr_{i2},\ hwdata_{i2},\ hgrant_{o2},\ clken_{i2},\ hwrite_{i3},\ htrans_{i3},\ hsize_{i3},\ hburst_{i3},\ hprot_{i3},\ hbusreq_{i3},\ hlock_{i3},\ haddr_{i3},\ hwdata_{i3},\ hgrant_{o3},\ clken_{i3},\ hwrite_{i4},\ htrans_{i4},\ hsize_{i4},\ hburst_{i4},\ hprot_{i4},\ hbusreq_{i4},\ hlock_{i4},\ haddr_{i4},\ hwdata_{i4},\ hgrant_{o4},\ clken_{i4},\ hwrite_{i5},\ htrans_{i5},\ hsize_{i5},\ hburst_{i5},\ hprot_{i5},\ hbusreq_{i5},\ hlock_{i5},\ haddr_{i5},\ hwdata_{i5},\ hgrant_{o5},\ clken_{i5},\ hgrant_{o0},\ hready_i,\ hresp_i,\ hrdata_i,\ hsplit_i,\ hrdata_o,\ hmaster_o,\ hmastlock_o,\ hwrite_o,\ haddr_o,\ hwdata_o)$

For the remainder of this dissertation, the above representation is denoted as **AHB**.

8.2 ARM7

8.2.1 ARM7 Specification

ARM7 is a 32-bit microprocessor from *Advanced RISC Machines* (ARM) [4, 33, 81]. It is based on the *Reduced Instruction Set Computer* (RISC) architecture. It has twenty eight instructions [1] and employs three stages of instruction pipeline processing.

Interface

An ARM7 processor has three bus lines (*PDataIn, PDataOut, PAddress*), four input control lines (*PnReset, PnWait, PnIRQ, PnFIQ*), and six output control lines (*PLock, PnRW, PnMreq, PSeq, PnBW, PnMode*). The descriptions for the interfaces of an ARM7 processor are presented in Table 8.2.

[1] The ARM7 specification defined in the data-sheet [4] has thirty three instructions. The ARM7 model described in this chapter only implemented twenty eight instructions. The five missing instructions are the co-processor instructions.

Name	Description
PnReset	is the reset signal. A LOW level input signal forces the processor to go to the default reset state.
PnWait	is the wait signal. A LOW level input signal stalls the processor.
PLock	is the lock signal. A HIGH level output signal indicates that the processor is performing a locked memory access.
PnRW	is the read or write signal. The processor is in the read cycle when the signal is LOW. Otherwise, it is in the write cycle.
PnMreq	is the memory request signal. When the signal is LOW, it indicates that the processor requires memory access.
PSeq	is the sequential address signal. When the signal is HIGH, the address of the next memory cycle relates to the last memory address
PnBW	is the signal to the external memory to indicate the length of data for the transfer. A HIGH signal indicates word length data, otherwise it is byte length data.
PnMode	is the signal to indicate the processor mode.
PnIRQ	is the fast interrupt signal.
PnFIQ	is the slow interrupt signal.
PBigend	is the signal to indicate the endian configuration. A HIGH signal indicates the processor treats the bytes in the memory in a big endian format
PDataIn	is the input data line. This input line allows the data to come into the processor.
PAddress	is the address line. This output line provides the memory address that the processor is accessing.
PDataOut	is the output data lines. This output lines provide the data to be transfered out of the processor.

Tab. 8.2: ARM7 Interfaces

Internal registers and Flags

The processor can be operated in six operating modes: user, FIQ, IRQ, supervisor, Abort, and undefined. Each operating mode has 16 active *General Purpose Registers* or GPR (R_0 – R_{15}) and one or two status registers, which are the *Current Program Status Register* (CPSR) and the *Saved Program Status Register* (SPSR). All operating modes have their own private registers: R_{13} and R_{14}. They share the use of other registers, except for the FIQ mode. The FIQ mode has five more private registers (R_8 – R_{12}). R_{13} is used to save the stack pointer, while R_{14} is used to store the link register. The user mode is the only mode which has only one status register (CPSR). It does not have the SPSR register. The register mapping for all modes is presented in Figure 8.3.

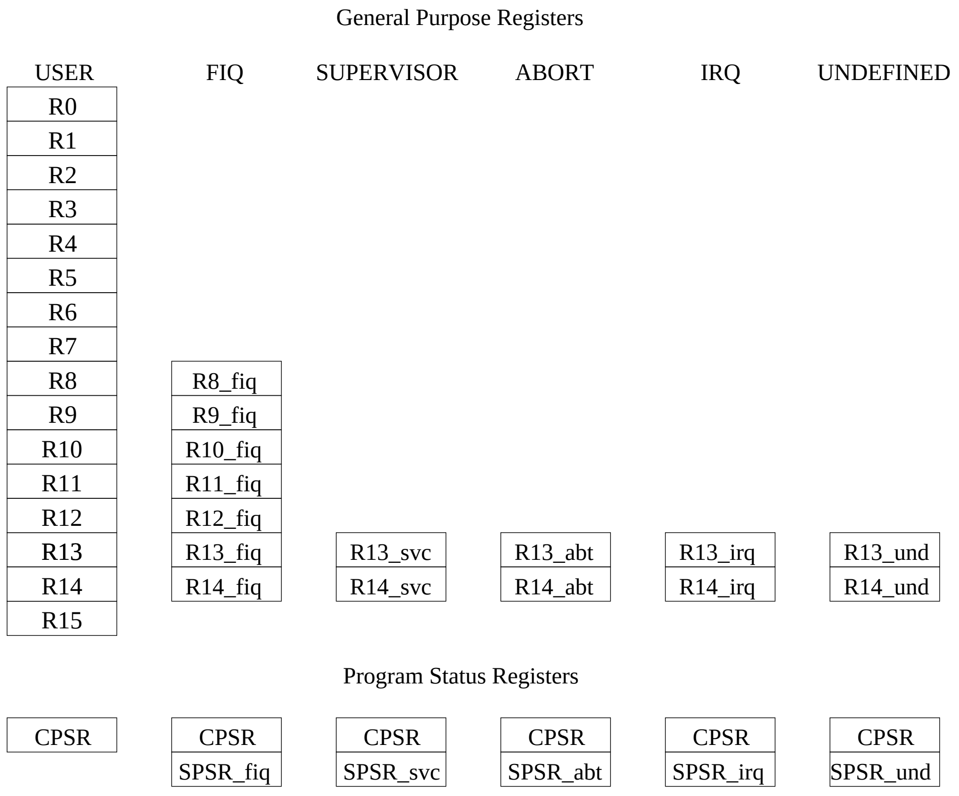

Fig. 8.3: ARM7 Registers Organisation

The CPSR stores information on the processor flags, interrupt disable bits, and the processor's operational mode (M). There are four flags: the negative flag (N), the zero flag (Z), the carry flag (C), and the overflow flag (O). These four flags can be changed as a result of executing an arithmetic or logical operation. The flags are used also to determine if the current instruction is to be executed. There are two interrupt status bits, one each for IRQ (I) and FIQ (F). When these bits are set then the corresponding interrupts are disabled. The I, F, and M are known as the processor's control bits. They can be changed only when the processor is not in the user mode. The format of the program's status register is presented in Figure 8.4.

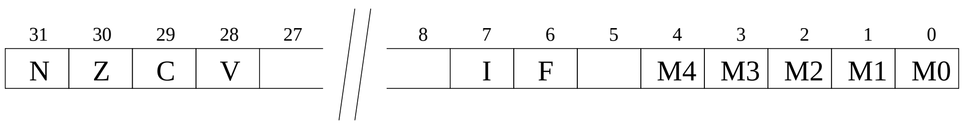

Fig. 8.4: Format of the Program Status Registers

Instruction Sets

Every instruction in ARM7 is 32-bits wide. The encoding of these instructions is presented in Figure 8.5. The last four bits (bit 28–31) of the instruction are used as the conditional prerequisite for the instruction to be executed. They represent the conditional status for N, Z, C, and V flags. If the corresponding flags in CPSR agreed with the conditions of the instruction, then the instruction is executed. Otherwise, the processor will perform a *no-operation* (NOP) process A summary of the conditional test is presented in Table 8.3

Code	Suffixes	Description
0000	EQ	Z is set
0001	NE	Z is clear
0010	CS	C is set
0011	CC	C is clear
0100	MI	N is set
0101	PL	N is clear
0110	VS	O is set
0111	VC	O is clear
1000	HI	C is set and Z is clear
1001	LS	C is clear or Z is set
1010	GE	N is equal to O
1011	LT	N is not equal to O
1100	GT	Z is clear, and N is equal to O
1101	LE	Z is clear, and N is not equal to O
1110	AL	ALWAYS execute instruction
1111	NV	NEVER execute instruction

Tab. 8.3: Instruction Condition Codes

The ARM7 processor has twenty eight instructions. Based on the type of operations and instruction cycles, these instructions can be categorised into eight groups: data processing, multiplication, branching, PSR data transfer, 31 29 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

a	Cond	0 0	I	Opcode	S	Rn	Rd	Operand 2			
b	Cond	0 0 0 0 0 0	A	S	Rd	Rn	Rs	1 0 0 1	Rm		
c	Cond	0 0 0 1 0	B	0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm		
d	Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
e	Cond	0 1 1	XXXXXXXXXXXXXXXXXXXX	I	XXXX						
f	Cond	1 0 0	P	U	S	W	L	Rn	Register List		
g	Cond	1 0 1	L	Offset							
h	Cond	1 1 0	P	U	N	W	L	Rn	CRd	CP#	Offset
i	Cond	1 1 1 0	CP Op	CRn	CRd	CP#	CP	0	CRm		
j	Cond	1 1 1 0	CP Op	L	CRn	Rd	CP#	CP	1	CRm	
k	Cond	1 1 1 1	Ignored								

a. Data Processing and PSR Transfer
b. Multiply
c. Single Data Swap
d. Single Data Transfer
e. Undefined
f. Block Data Transfer
g. Branch
h&i. Coprocessor Instruction
j. Software Interrupt

Fig. 8.5: ARM Instruction Set Summary

single data transfer, multiple data transfer, data swap, and software interrupt. All single cycle arithmetic instructions are gathered in the data processing group. This group contains sixteen instructions, ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN, AND, ORR, EOR, MOV, MVN, BIC, TST, and TEQ. The multiplication instructions (MUL, MLA) are separated from the data processing group as they may take more than one clock cycle to complete. The processor has two branching instructions B and BL. It has two instructions to manipulate the CPSR register, one for reading (MSR) and one for writing (MRS). Both single and multiple data transfer groups contain two instructions, one for reading from memory and one for writing to memory. The single data instructions are LDR and STR. The multiple data transfer instructions are LDM and STM. The last two instructions are the single data swap SWP and the software interrupt SWI. A summary of these instructions is described in Table 8.4.

Based on the type of memory transfer, the clock cycles in ARM7 are categorised into four classes: non-sequential (N), sequential (S), Internal (I), and Coprocessor (C). In the N-cycle, the processor requests a transfer to or from an address that is unrelated to the address used in the previous cycle. While in the S-cycle, the address of the data transfer is related to the previous one. In the I-cycle, the processor does not require data transfer. Similar to this, the C-cycle does not require data transfer, but it uses the bus to communicate with the co-processor. These processor cycle classes are defined by the output signals of *PnMreq* and *PSeq*. The corresponding relations between these signals and the memory cycle type are shown in Table 8.5.

The ARM7 processor features a three-stage pipeline architecture. Typically in

Mnemonic	Instruction	Description
ADC	Add with carry	Rd $\leftarrow$ Rn + Op2 + Carry
ADD	Add	Rd $\leftarrow$ Rn + Op2
AND	Logical and	Rd $\leftarrow$ Rn $\wedge$ Op2
B	Branch	R15 $\leftarrow$ #address
BIC	Bit clear	Rd $\leftarrow$ Rn $\wedge\neg$ Op2
BL	Branch with link	R14 $\leftarrow$ R15, R15 $\leftarrow$ #address
CMN	as ADD but result not written	CPSR flags $\leftarrow$ Rn + Op2
CMP	as SUB but result not written	CPSR flags $\leftarrow$ Rn - Op2
EOR	Logical exclusive or	Rd $\leftarrow$ Rn $\oplus$ Op2
LDM	Load multiple registers	Stack manipulation
LDR	Load register from memory	Rd $\leftarrow$ #address
MLA	Multiply accumulate	Rd $\leftarrow$ (Rm * Rs) + Rn
MOV	Move register or constant	Rd $\leftarrow$ Op2
MRS	Move PSR to register	Rn $\leftarrow$ PSR
MSR	Move register to PSR	PSR $\leftarrow$ Rm
MUL	Multiply	Rd $\leftarrow$ (Rm * Rs)
MVN	Move negated register	Rd $\leftarrow$ $\neg$ Op2
ORR	Logical or	Rd $\leftarrow$ Rn $\vee$ Op2
RSB	reversed subtraction	Rd $\leftarrow$ Op2 - Rn
RSC	reversed subtraction with carry	Rd $\leftarrow$ Op2 - Rn - 1 + Carry
SBC	subtraction with carry	Rd $\leftarrow$ Rn - Op2 - 1 + Carry
STM	store multiple registers	Stack manipulation
STR	Store register to memory	#address $\leftarrow$ Rd
SUB	subtraction	Rd $\leftarrow$ Rn - Op2
SWI	Software interrupt	O/S call
SWP	Swap register with memory	Rd $\leftarrow$ Rn, Rn $\leftarrow$ Rd
TEQ	as EOR but result not written	CPSR flags $\leftarrow$ Rn EOR Op2
TST	as AND but result not written	CPSR flags $\leftarrow$ Rn AND Op2

Tab. 8.4: Instructions Summary

nMREQ	SEQ	Cycle type
0	0	N (Non-Sequential)
0	1	S (Sequential)
1	0	I (Idle)
1	1	C (Co-Processor)

Tab. 8.5: Memory Cycle Type

one clock cycle, one instruction is fetched and stored in the last pipeline queue, one instruction in the second pipeline queue is decoded, and one instruction in the top pipeline queue is executed. The number of clock cycles used to complete the execution process varies depending on the type of instruction. For example a typical data processing instruction can be completed in one S-cycle. But, when R_{15} is used to store the result of the operation, the elapsed time is increased by two clock cycles (one S-cycle and one N-cycle). The summary of elapsed time of ARM7 instructions is presented in Table 8.6. The value of n is determined by the number of words transfered. The value of m is determined by the number of cycles required by the multiplier algorithm. For example, multiplication by any number between 2^{2m-3} and (2^{2m-1} - 1) and (1 < m > 16) takes 1S + mI cycles. Multiplication by 0 or 1 uses 1S + 1I cycles. Multiplication by any number greater or equal 2^{29} uses 1S + 16I cycles.

Instruction	Cycle count	Additional	
Data Processing	1S	+ 1I	for SHIFT(Rs)
		+ 1S + 1N	if R15 written
MSR, MRS	1S		
MUL, MLA	1S + mI		
LDR	1S + 1N + 1I	+ 1S + 1N	if R15 loaded
STR	2N		
LDM	nS + 1N + 1I	+ 1S + 1N	if R15 loaded
STM	(n-1)S + 2N		
SWP	1S + 2N + 1I		
B, BL, SWI	2S + 1N		

Tab. 8.6: ARM7 Instruction Cycle Summary

8.2.2 Formal Model

The ARM7 processor is modelled in LISP (ACL2). The approach is similar to the one used to model the processor module in SIP. Both models use the *state* function to represent their internal condition. The FSM of ARM7 has more states than the number of states in the SIP FSM. This is because ARM7 has more functionality. Some of the instructions perform iterative processes, such as the instructions for multiplication and multiple data transfer. Even though the ARM7 FSM is more complex than the processor in SIP, the approach used to model each instruction cycle remains the same. Every cycle of the instruction is modelled as a function. The evaluation of this instruction cycle updates the value of the *state* function.

Basic components

The *state* of ARM7 defines the internal state of the processor. It contains a list of components of the processor, such as registers and flags. In total, *state* contains a list of sixteen components.

Seven of them are the output signals from the processor. These signals are the *pmreq* (memory request), *pseq* (sequential indicator), *prw* (memory read/write), *pbw* (byte/word data), *plock* (lock request), *padd* (memory address), and *pdo* (data-out). Detailed explanations of these signals are presented in Table 8.2.

Three of the components are the pipeline registers: *pp0, pp1*, and *pp2*. *pp0* is the top pipeline register. *pp1* is the second pipeline register. *pp2* is the last pipeline register. In one clock cycle, the processor perform fetch, decode and execute operations simultaneously. The processor executes instructions in *pp0*. At the same time, it decodes instructions in *pp1* so that all controls are ready when the processor completes the execution. Meanwhile, it fetches a new instruction from external memory and stores the data in *pp2*.

Registers in ARM7 are represented using two variables: *preg* and *pcspr*. *preg* contains the GPRs registers. *pcspr* contains all status registers. The registers are organised based on the processor modes. The first sixteen locations are allocated for processor mode 0 (user). The next seven locations are allocated for mode 1 (FIQ). The last eight locations are allocated for modes 2 to 5 (supervisor, abort, IRQ, and undefined), with two memory locations allocated for each mode. The CPSR and SPSRs registers are stored in the *pcspr*. The CPSR register is placed in the first list and followed by the SPSRs for mode 1 to mode 5.

Finally, the last four components are the auxiliary components. They are *pnreset, pfsm, ptempPC*, and *pvar*. *pnreset* is the flag that indicates that the processor is in a reset state. *pfsm* is the processor's state machine. *ptempPC* is the temporary PC register. *pvar* is the general purpose variable and is used as a counter variable in multiplication and multiple data transfer instructions. It is also a temporary variable when performing swap operations.

Definition 38 (ACL2 ARM7's state: state)

$$state(preg\ pcspr\ pnreset\ pfsm\ pp0\ pp1\ pp2\ pnmreq\ pseq\ pnrw\ pnbw\ plock\ padd\ pdo\ ptempPC\ pvar) \overset{def}{=} list\ preg\ pcspr\ pnreset\ pfsm\ pp0\ pp1\ pp2\ pnmreq\ pseq\ pnrw\ pnbw\ plock\ padd\ pdo\ ptempPC\ pvar$$

Similar to SIP, every variable in *state* has its own accessor function. Overall there are seventeen accessor functions, one for each variable. Using these accessor functions, the value of each variable can be obtained. For example the accessor function *PReg* applied to *state* results in the memory component of GPRs.

The Processor's Instruction Sets

In this section we briefly describe each instruction cycle. The modelling approach of the instructions is exactly the same as the one used to model SIP. Based on ARM7 types of operation, these instructions can be grouped into seven classes. The classification of the instructions is described in Table 8.7.

Class	Instruction
Branch	B, BL
Data processing	ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN, AND, ORR, EOR, MOV, MVN, BIC, TST, TEQ
PSR transfer	MRS, MSR
Multiply	MUL, MLA
Data transfer	LDR, STR, LDM, STM
Single data swap	SWP
Software Interrupt	SWI

Tab. 8.7: ARM7 Instruction Classes

• No Operation

All instructions in ARM7 processor are conditionally executed. An instruction is executed only when it passes the execution condition. Otherwise, a (MOV R_0 R_0) operation is performed. In our model, this instruction is replaced by a *no operation* (NOP) instruction. The NOP process only fetches a new instruction and stores it in the pipeline register.

$$
\begin{array}{llll}
NOP_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
NOP_h^1 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}
$$

• Branching Instructions: B and BL

Two instructions fall into the branch instruction group: regular branch (B) and subroutine branch (BL). A subroutine branch is capable of returning to the next instruction after the branch instruction, while the regular branch does not have this option. When the processor executes BL, the address of the instruction stored in *pp1* (pipeline register 1) is copied to R_{14}. The program can return from a subroutine by restoring the PC data from R_{14} by using a move instruction (MOV PC,R_{14}).

A branch instruction uses the first 24 bits of the instruction code as the offset value. This data is shifted by two bits to create a +/-32Mbyte offset. The offset is added to the address of the current instruction (*pp0*). The program counter (PC) which is eight bytes ahead of *pp0* is used as the reference address. The branch instruction is completed in three clock cycles.

$$
\begin{array}{lllll}
branch_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & & & \text{L} \rightarrow ([\text{R}_{14}] \leftarrow [\text{R}_{15}]\text{-4}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
 & & \text{ptempPC} & \leftarrow & \text{PC-8+offset} \\
branch_h^1 & : & \text{pfsm} & \leftarrow & 8 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}
$$

In the first cycle, a new instruction is being fetched and stored in the pipeline. A new address is prepared and stored in the temporary PC register (*ptempPC*). When a branch with a link instruction is executed (L), the return address is saved in R_{14}. The saved address is the PC adjusted by four bytes. The next two cycles are used to flush the instructions from the pipeline and replace them with instructions denoted by the branching address.

$$\begin{array}{lcllll}
flush_l^2 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{ptempPC}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{ptempPC} \\
 & & \text{pnrw,pnmreq,plock} & \leftarrow & \text{nil} \\
 & & \text{pseq,pnbw,pnreset} & \leftarrow & \text{t} \\
flush_h^2 & : & \text{pfsm} & \leftarrow & 9 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}$$

The second cycle of a branch instruction is the first pipeline flush (flush^2). In this stage, a new instruction from a new address is fetched and stored in the pipeline. The old instruction is flushed and not executed.

$$\begin{array}{lcllll}
flush_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,plock} & \leftarrow & \text{nil} \\
 & & \text{pseq,pnbw,pnreset} & \leftarrow & \text{t} \\
flush_h^1 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}$$

The third cycle is the last pipeline flush cycle (flush^1). It performs a similar process to the first pipeline flush cycle. These two flush cycles are also used by instructions which use R$_{15}$ as the destination register and the content of R$_{15}$ is updated with a new one.

- **Data Processing**

There are sixteen instructions for performing arithmetic and logical operations. ADD, ADC, SUB, SBC, RSB, RSC, CMP, and CMN are classified as the arithmetic instructions. The rest are classified as the logical instructions. These eight instructions are AND, ORR, EOR, MOV, MVN, BIC, TST, and TEQ. The descriptions of each instruction are presented in Table 8.4. Two instructions from each category do not have a destination register and do not write the result in any register. These instructions are CMP, CMN, TST, and TEQ. The other instructions use the destination register as the target register to store the result of the data processing operations. When the S bit in the instruction is set, the result of the data processing operation is used to update the V, C, Z, and N flags in the CSPR. In general the data processing instructions complete the process in a single cycle. But, it is possible for the operations to be completed in two or three cycles.

$$
\begin{array}{lllll}
\mathit{dataproc}_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & & & (\text{updateRd} \wedge \neg\text{shift\_Rs}) \rightarrow \\
 & & & & \qquad ([\text{Rd}] \leftarrow \text{process}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pseq} & \leftarrow & \text{shift\_Rs} \\
 & & \text{pnrw,pnmreq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
 & & \text{pcspr} & \leftarrow & (\neg\text{shift\_Rs} \wedge \text{S}) \rightarrow \text{update\_cpsr} \\
\mathit{dataproc}_h^1 & : & \text{pfsm} & \leftarrow & \text{shift\_Rs} \rightarrow 15 \mid 1 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}
$$

Almost all data processing instructions complete their operations in the first cycle. Externally, a new instruction is being fetched and stored in the pipeline.

$$
\begin{array}{lllll}
\mathit{dataproc}_l^2 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & & & \text{updateRd} \rightarrow ([\text{Rd}] \leftarrow \text{process}) \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
 & & \text{pcspr} & \leftarrow & \text{S} \rightarrow \text{update\_cpsr} \\
\mathit{dataproc}_h^2 & : & \text{pfsm} & \leftarrow & 1
\end{array}
$$

When the second register is shifted, one more clock cycle is needed to complete the process. In this case, the first cycle is only used to fetch a new instruction and shift the value of the second register. The arithmetic or logical operations are completed in the second cycle. Flags are updated after the arithmetic or logical operations have been completed.

$$
\begin{array}{lllll}
\mathit{PC\_update}_l^x & : & \text{ptempPC} & \leftarrow & \text{process\_data} \\
\mathit{PC\_update}_h^x & : & \text{pfsm} & \leftarrow & 8
\end{array}
$$

When the instruction uses the PC register (R_{15}) as the destination register, two additional cycles are added to flush existing instructions in the pipelines. The flushing processes are similar to the second and third cycles of branching instructions (flush^2 and flush^1). The new address/data for the PC register is stored in *ptempPC* which is loaded to the output address port.

- **Multiplication**

ARM7 has two multiplication instructions, MUL and MLA. MUL operates on Rm and Rs registers while MLA has an additional register (Rn) which is accumulated with the result of Rm and Rs. Similar to the data processing processes, when the S bit is set then the status flags (N,C,Z, and V) from CSPR are updated.

$MULT_l^1$	:	preg	$\leftarrow$	$([\mathrm{R}_{15}] \leftarrow$ padd+4)
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+4
		pnrw,pnmreq,plock	$\leftarrow$	nil
		pseq,pnbw,pnreset	$\leftarrow$	t
$MULT_h^1$	:	pfsm	$\leftarrow$	([Rs] = 0,1) $\rightarrow$ 11 | 10
		pp2	$\leftarrow$	di
		pvar	$\leftarrow$	m

In the first cycle, a new instruction is fetched and stored in the pipeline register. In this cycle, the processor determines the value of the multiplicand. If the value is zero or one then in the next cycle it performs $MULT^3$, otherwise it goes to $MULT_l^2$. In ARM7, the number of cycles needed by the multiplication algorithm to complete the operation is determined by the value of the multiplicand. At most the instruction uses seventeen clock cycles to complete its execution. It is possible for the algorithm to complete the process before the maximum allocated time. The operation is modelled by setting the maximum number of iterations *m*. The iteration value is stored in the temporary variable *pvar*.

$MULT_l^2$	:	preg	$\leftarrow$	(pvar = m) $\rightarrow$ $([\mathrm{R}_{15}] \leftarrow$ padd+4)
		padd	$\leftarrow$	(pvar = m) $\rightarrow$ padd+4
		pnrw,pseq,plock	$\leftarrow$	nil
		pnmreq,pnbw,pnreset	$\leftarrow$	t
$MULT_h^2$	:	pfsm	$\leftarrow$	((pvar-1) = 1) $\rightarrow$ 11 | 10
		pvar	$\leftarrow$	(pvar-1)

$MULT^2$ is used as the internal iteration state of the processor. Each iteration reduces the value of *pvar* by one. The process remains in this state until the value of *pvar* reaches one. This indicates that the multiplication process will be completed in the next cycle. The processor needs to move to a new state of $MULT^3$.

$MULT_l^3$	:	preg	$\leftarrow$	([Rs] = 0,1) $\rightarrow$ $([\mathrm{R}_{15}] \leftarrow$ padd+4)
				[Rd] $\leftarrow$ (A $\rightarrow$ ([Rm]*[Rs])
				| ([Rm]*[Rs]+[Rn]))
		padd	$\leftarrow$	([Rs] = 0,1) $\rightarrow$ padd+4
		pnrw,pnmreq,plock	$\leftarrow$	nil
		pseq,pnbw,pnreset	$\leftarrow$	t
$MULT_h^3$	:	pfsm	$\leftarrow$	1

$MULT^3$ is the last state or cycle of the multiplication operations. In this cycle, the result of the multiplication is calculated and stored in the destination register. MUL and MLA are not allowed to write the result in the PC register.

• Data Transfer

There are four instructions for data transfer, LDR, STR, LDM, STM. LDR and STR are the single load and store data transfer, while LDM and STM are the block load and store data transfer. In general, single data transfer operations are block data transfer operations for transferring single data. The choice of instruction to do write or read is provided by the L bit. The processor performs a load transfer when the L bit is HIGH and a store transfer when it is LOW. The U bit determines the direction of the offset. When it is HIGH the offset is added to the base address and when it is LOW the offset is subtracted from the base address. When the W bit is set to HIGH then the new updated address is written back to the base address (Rn). If the P bit is LOW then the indexed offset of the address is used for the transfer. Otherwise, it uses the pre-indexed value. In the single data transfer, there is an option to do either a byte or word transfer, while block data transfers always occur at the word level.

$load_l^1/store_l^1$	:	preg	$\leftarrow$	([R_{15}] $\leftarrow$ padd+4)
		pp0	$\leftarrow$	pp1
		pp1	$\leftarrow$	pp2
		padd	$\leftarrow$	padd+4
		pnrw,pnmreq,pseq,plock	$\leftarrow$	nil
		pnbw,pnreset	$\leftarrow$	t
$load_h^1/store_l^1$	:	pfsm	$\leftarrow$	load $\rightarrow$ 3
				store $\rightarrow$ 2
		pp2	$\leftarrow$	di
		ptempPC	$\leftarrow$	LDR/STR $\rightarrow$ (Rn or offset)
				LDM/STM $\rightarrow$ adjusted_Rn
		pvar	$\leftarrow$	LDR/STR $\rightarrow$ 1
				LDM/STM $\rightarrow$ itt

In the first load/store cycle, a new instruction is fetched and stored in the pipeline. Meanwhile, a new address into which the transfer will be made is prepared and stored in the temporary PC (*ptempPC*). Finally the total amount of data that will be fetched is stored in *pvar*. In a single data load transfer, only one data is fetched or stored. In this case, *pvar* is stored with the value of one. *pvar* in the block data transfer stores the number of transfers that will be made. The single load and store can complete the operations in a minimum of three clock cycles and two cycles respectively. While the block data transfer uses *(itt - 1)* cycles more, where *itt* is the amount of data being loaded.

$load_l^2$	:	preg	$\leftarrow$	([R_{15}] $\leftarrow$ ptempPC)
				(pvar = itt) $\rightarrow$ ([R_{14}] $\leftarrow$ padd+4)
		padd	$\leftarrow$	ptempPC
		pnrw,pnmreq,plock	$\leftarrow$	nil
		pseq,pnreset	$\leftarrow$	t
		pnbw	$\leftarrow$	LDR $\rightarrow$ B
				LDM $\rightarrow$ t
$load_h^2$	:	preg	$\leftarrow$	([R_{list}] $\leftarrow$ di)
		pfsm	$\leftarrow$	(pvar > 1) $\rightarrow$ 3 | 4
		ptempPC	$\leftarrow$	LDM $\rightarrow$ adjusted_Rn
		pvar	$\leftarrow$	pvar - 1

In the first half of the second load cycle ($load_l^2$), the new address is posted in the address port. If this is the first loop cycle, then the return address is stored in R_{14}. In the second half cycle ($load_h^2$), the data from the memory is read and stored according to the register list. If all the data is fetched then the operation will go to the last load sequence ($load^3$). Otherwise a new address is prepared for the next load cycle and the number of loops to be made will be decreased by one. LDR can complete the operation in a minimum of three clock cycles. The STM uses (loop - 1) cycles more than LDR, where n is the number of data being loaded.

$load_l^3$	:	preg	$\leftarrow$	$([R_{15}] \leftarrow [R_{14}])$
		padd	$\leftarrow$	$[R_{14}]$
		pnrw,pnmreq,plock	$\leftarrow$	nil
		pseq	$\leftarrow$	$\neg(\text{Rd} = \text{PC})$
		pnbw,pnreset	$\leftarrow$	t
$load_h^3$	:	pfsm	$\leftarrow$	1

In the final load cycle the return address is restored. If R_{15} is used as the destination register, the two cycle flush sequence will follow the $load^3$ operation. In addition, the *PC_update* sequence will be executed. A new address is then loaded to ptempPC.

$store_l^2$	:	preg	$\leftarrow$	$([R_{15}] \leftarrow \text{ptempPC})$
				$(\text{pvar} = \text{itt}) \rightarrow ([R_{14}] \leftarrow \text{padd}+4)$
		padd	$\leftarrow$	ptempPC
		pdo	$\leftarrow$	$[R_{list}]$
		pseq,pnmreq,plock	$\leftarrow$	nil
		pnrw,pnreset	$\leftarrow$	t
		pnbw	$\leftarrow$	STR $\rightarrow$ B
				STM $\rightarrow$ t
$store_h^2$	:	pfsm*	$\leftarrow$	$(\text{pvar} > 1) \rightarrow 2 \mid 1$
		ptempPC	$\leftarrow$	STM $\rightarrow$ adjusted_Rn
		pvar	$\leftarrow$	pvar - 1

In the first half of second store cycle ($store_l^2$), data from the register list is sent to the data-out port and a new address is sent to the address port. If this operation is the first loop cycle, then the return address is stored in R_{14}. After all operations are completed the return address will be adjusted in the beginning of the next clock cycle.

- **PSR transfer**

ARM7 provides two instructions to manipulate the PSR registers. The MRS instruction allows the contents of the PSR register to be transfered to a GPR. The MSR instruction allows the contents of a GPR or an immediate data to be stored to the CPSR or SPSR. The choice between CPSR or SPSR is selected by the P bit. If the P is HIGH then the operation will manipulate SPSR, if it is LOW the CPSR will be manipulated.

$$\begin{array}{lcl c l}
PSR_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & & & \text{MRS} \rightarrow ([\text{Rd}] \leftarrow (\text{P} \rightarrow \text{SPSR} \mid \text{CPSR})) \\
 & & & & \text{MSR} \rightarrow (\text{P} \rightarrow ([\text{SPSR}] \leftarrow (\text{Rm or Imm})) \\
 & & & & \qquad\qquad \mid \ ([\text{CPSR}] \leftarrow (\text{Rm or Imm}))) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
PSR_h^1 & : & \text{pfsm} & \leftarrow & 1 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}$$

While the processor performs a PSR data transfer, a new instruction is fetched and stored in the pipeline register. When the MRS instruction is executed the data from CPSR or SPSR is copied to Rd in the GPR. Meanwhile, the MSR instruction moves the content of Rm or an immediate value to either CPSR or SPSR. PSR instructions use one clock cycle to complete the data transfer.

• Single Data Swap

The data swap instruction (SWP) is used to swap data between the internal register and the external memory. The instruction performs a read operation which is followed by a write operation. The operations are done in a lock mode, which is a priority operation that can not be interrupted until it is completed. The swap address is determined by the value of Rn. The processor stores the content of Rm in the swap address and stores the external memory value in Rd. The swap instruction is executed in four clock cycles.

$$\begin{array}{lcl c l}
swap_l^1 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{padd+4}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
swap_h^1 & : & \text{pfsm} & \leftarrow & 5 \\
 & & \text{pp2} & \leftarrow & \text{di} \\
 & & \text{ptempPC} & \leftarrow & [\text{Rn}]
\end{array}$$

In the first cycle, a new instruction is fetched and stored in the pipeline register. The swap address is read from Rn and stored in *ptempPC*.

$$\begin{array}{lcl c l}
swap_l^2 & : & \text{preg} & \leftarrow & ([\text{R}_{15}] \leftarrow \text{Rn}) \\
 & & & & ([\text{R}_{14}] \leftarrow \text{padd+4}) \\
 & & \text{padd} & \leftarrow & \text{Rn} \\
 & & \text{pnrw,pnmreq,pseq} & \leftarrow & \text{nil} \\
 & & \text{plock,pnreset} & \leftarrow & \text{t} \\
 & & \text{pnbw} & \leftarrow & \text{B} \\
swap_h^2 & : & \text{pfsm} & \leftarrow & 6 \\
 & & \text{pvar} & \leftarrow & \text{di}
\end{array}$$

In the first half of the second swap cycle, the swap address is assigned to the address port and the return address is stored in R_{14}. In the second half cycle data from the external memory is read and stored in the temporary variable pvar.

$$\begin{array}{lcllcl}
swap^3_l & : & \text{pdo} & \leftarrow & [\text{Rm}] \\
 & & \text{pseq} & \leftarrow & \text{nil} \\
 & & \text{pnrw,pnmreq,plock,pnreset} & \leftarrow & \text{t} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{B} \\
swap^3_h & : & \text{pfsm} & \leftarrow & 7
\end{array}$$

In the third cycle ($swap^3$), data from Rm is sent to the data-out port and stored in the external memory.

$$\begin{array}{lcllcl}
swap^4_l & : & \text{preg} & \leftarrow & ([R_{15}] \leftarrow [R_{14}]) \\
 & & & & ([\text{Rd}] \leftarrow \text{pvar}) \\
 & & \text{padd} & \leftarrow & [R_{14}] \\
 & & \text{pnrw,pnmreq,plock} & \leftarrow & \text{nil} \\
 & & \text{pseq,pnbw,pnreset} & \leftarrow & \text{t} \\
swap^4_h & : & \text{pfsm} & \leftarrow & 1
\end{array}$$

Finally, in the last swap cycle ($swap^4$) the content of the old external memory is written in the destination register. The address is restored to the value saved in R_{14}.

• Software Interrupt

The software interrupt instruction is used to enter the supervisor mode. The instruction causes a software interrupt trap which changes the operating mode. The CPSR is saved in SPSR_svc. The PC is forced to have 0x08H and the old PC address is saved in R_{14}. The processor can return to its previous operating mode by restoring the PC and CPSR values.

$$\begin{array}{lcllcl}
SWI^1_l & : & \text{preg} & \leftarrow & ([R_{15}] \leftarrow \text{padd+4}) \\
 & & & & ([R_{14}] \leftarrow \text{padd+4}) \\
 & & \text{pp0} & \leftarrow & \text{pp1} \\
 & & \text{pp1} & \leftarrow & \text{pp2} \\
 & & \text{padd} & \leftarrow & \text{padd+4} \\
 & & \text{pnrw,pnmreq,pseq,plock} & \leftarrow & \text{nil} \\
 & & \text{pnbw,pnreset} & \leftarrow & \text{t} \\
 & & \text{pcpsr} & \leftarrow & \text{SPSR}_{svc} \leftarrow \text{CSPR} \\
 & & \text{ptempPC} & \leftarrow & 8 \\
SWI^1_h & : & \text{pfsm} & \leftarrow & 8 \\
 & & \text{pp2} & \leftarrow & \text{di}
\end{array}$$

SWI stores the return address at R_{14} and sets the new address to 0x08H. After that it goes to two cycle flush sequences and loads new instructions in the supervisory mode.

Processor's Execution Model

The top level ARM7 processor function is the single cycle behaviour function (*ARM7execute*). The function takes five arguments: the reset signal (*PnReset*), the wait signal (*PnWait*), the interrupt signals (*PnFRQ, PnIRQ*), the data-in (*PDataIn*), and the processor's state (*Pstate*).

The *ARM7execute* function implements four exceptions. A fixed priority system controls the exceptions, with the highest priority being the reset signal. When the processor recieves a reset signal then the processor goes into a default reset state. The next priority is for the interrupt signals. There are two external interrupt signals: the fast interrupt (*PnFRQ*) and the slow interrupt request signal (*PnIRQ*). A fast interrupt request signal has higher priority than the slow interrupt request signal. These two execptions are executed when the F and I flags in the CPSR are enabled. The last execption is the software interrupt.

Two conditions determine whether the processor executes the instruction in the pipeline register. First, the processor must not receive a wait signal or a no grant signal. Otherwise, the internal state of the processor remains unchanged. Second, the condition of the instruction has to be true. The top four bits of the instruction have to satisfy the conditions described in Table 8.3. If the result of the condition is false, the current instruction is ignored and a no operation is executed.

8.2.3 Wrapper

The ARM7 processor is connected to an AMBA bus as the default master. Unfortunately, ARM7 external specifications do not comply with AMBA specifications. The ARM7 processor uses a two-clock cycle operation in completing the process. In the first cycle, control signals are sent by the processor to request external data. If it is granted, then the address is fetched in the second cycle and data will be ready in the same cycle. The AMBA bus protocol uses a three clock cycle operation in completing data transfer. Control signals are expected in the first clock cycle. Then in the second cycle the memory address is fetched by the slave. Data comes in the final clock cycle.

A wrapper is needed to interface the communication and data transfer between the ARM7 processor and the AMBA bus protocol [6, 8]. Input signals from AHB to ARM7 are routed through the wrapper and output signals are also routed through the wrapper before they are driven to AHB. The wrapper enables the ARM7 processor to satisfy the AHB master requirements. This wrapper was modelled in SMVL.

The basic ARM7 wrapper is made up of three blocks: wrap-burst, wrap-lock, and wrap-master. The wrap-burst module is responsible for generating AHB address (HAddr) and control signals during burst transfers. The control signals of a wrap-burst are: the size of data (HSize), the read/write (HWrite), and the type of transfer (HTrans) signals.

HWrite and HSize signals are buffered signals from the processor. HWrite is generated from the read/write (PnRW) signal of ARM7. The HSize signal is used to indicate the size of data from 8 bits up to 1024 bits. ARM7 is capable of performing an 8 bit or a 32 bit data transfer. The selection of bit sizes is shown by the PnBW signal. AMBA allows data transfer from size 8 up to 1024 bits which is indicated by a 3 bit HSize signal. The wrapper uses two sizes: *001* and *010*. *001* indicates a byte data transfer; while *010* is used for a 32 bit word data transfer.

There are three types of HTrans signals generated by wrap-burst: *idle*, *seq*, and *nonseq*. It is mainly generated from the PnMreq and PSeq control signals from the processor. When the processor performs a sequential cycle then the wrapper generates *seq* on HTrans. The only exception is that when the processor's sequential cycle follows an internal cycle then a *nonseq* is generated on HTrans. A non-sequential cycle on the processor also generates a *nonseq* on HTrans. When the processor is performing an internal cycle or a co-processor cycle, an *idle* is generated on HTrans.

The HAddr data comes from either the processor's output address or internal address incrementer. In general, the address from the processor is running one cycle ahead of the bus. When the processor performs a non-sequential cycle, a new address which is not related to a previous transfer is generated by the processor. An Idle cycle is asserted on the AHB to allow the new address to be sampled by the wrapper. In this cycle, the processor's clock is stopped to allow the wrapper to synchronise with the processor whilst the processor preserves its internal state. In the next cycle, the address sampled by the wrapper is used for the HAddr. If the following cycle is a sequential cycle then the internal address incrementer is used to generate the HAddr data. This process eliminates the need for the wrapper to assert an idle cycle.

The wrap-lock module generates the HLock signal. The HLock signal is only set when the processor executes a SWAP instruction. The module monitors all instructions read by the processor. It detects when the processor is about to perform a SWAP instruction and checks that the instruction is being executed when it exits the pipeline registers.

The wrap-master module controls the bus master interface to the AHB. The module is used to determine when the wrapper is granted the bus and when it can drive the address, data, and control outputs. It is responsible for generating the HBurst and HBusreq signals. The HBurst output is held at 001 to indicate that the processor only performs burst operations of unspecified length. The HBusreq output is held HIGH to show that the processor is always requesting access to the bus.

8.3 Summary

This chapter describes the specification and formalism of an ARM platform. It contains the AMBA AHB bus protocol and ARM7 processor. The bus protocol is defined in Verilog. It is capable of handling five master modules and one slave module. ARM7 is defined in ACL2. A wrapper is designed to interface the communication protocol between the processor and the AMBA AHB bus. These two modules are the main components of the ARM platform. Components are added to the platform to create an application or an application specific platform.

Chapter 9

The ARM Formal Verification Platform

In this chapter, we explain the development of a verification platform for the system described in Chapter 7. The platform is based on specifications for the AMBA AHB bus protocol and the ARM7 processor. Then, it is used to develop platform specifications for two AHB masters.

The development consists of three stages. First, the generic properties of the platform which are based only on RAPIER's AMBA protocol properties are established. Second, the properties of the processor are developed. Third, the bus arbiter and the processor are integrated by combining their properties. The result of this combination is used to define the specifications for the remaining components. These specifications are the test-benches to define the components' compatibilities with the system.

The generic properties of the system are obtained by proving properties about the components by describing the functional behaviour of the system with relevant input/output scenarios or sequences. The basic idea is to make the proof reusable by finding and defining the most general or weakest environmental constraints required to obtain the properties. These properties are used as the behavioural representation of the components.

In Section 9.1, the development of the generic AMBA AHB properties will be described and the constraints needed to make the bus protocol perform as specified will be defined. The analysis of the properties of ARM7 will be described in Section 9.2. Then in Section 9.3, the development of the application specific platform will be described. Finally in Section 9.4, a summary of this chapter is presented.

9.1 AMBA AHB Properties

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

All SMV verifications were performed using a Linux machine with an Intel Xeon 2.4GHz processor with 3G RAM.

9.1.1 Basic Operational Conditions and Properties

From the RAPIER documentation [65], 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, requests to access the bus will be ignored and no grant signal can be assigned to it. To achieve maximum coverage of all master 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 is formulated by defining that the reset signal is active for at least one cycle and no reset is applied afterwards. Analysis is only performed on the behaviour of the model after the system is reset and all masters are active. This constraint is defined in *Assumption 1.*

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

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. The SMV code is shown below:

```
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, bus ownership is granted to m_1 [7]. If the default master (m_1) is in split mode, the bus is granted to the 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 GX(grant_m0 | grant_m1 | grant_m2 |
              grant_m3 | grant_m4 | grant_m5);
using SMV_Assumption1 prove SMV_Theorem1;

SMV_Theorem2:
    assert GX(~(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, modules, and assumptions are used in the verification. The modules and assumptions become the antecedents and the properties being proved become the conclusions of implications in HOL. For example, SMV_Theorem1 and SMV_Theorem2 are imported into the HOL environment by using the command *get_smv_theorem*. The HOL theorem is:

HOL_Theorem:

$$
\begin{array}{l}
(\text{AHB} \wedge \text{Reset} \wedge \text{XG}\ \neg\text{Reset} \wedge \\
\quad \text{G}(\text{clocken\_m}_1 \wedge \text{clocken\_m}_2 \wedge \text{clocken\_m}_3 \wedge \text{clocken\_m}_4)) \\
\rightarrow \\
(\text{GX}(\text{grant\_m}_0 \vee \text{grant\_m}_1 \vee \text{grant\_m}_2 \vee \text{grant\_m}_3 \vee \text{grant\_m}_4 \vee \text{grant\_m}_5) \wedge \\
\quad \text{GX}(\neg(\text{grant\_m}_0 \wedge \text{grant\_m}_1)\wedge \\
\qquad \neg(\text{grant\_m}_0 \wedge \text{grant\_m}_2)\wedge \\
\qquad \ldots \\
\qquad \neg(\text{grant\_m}_4 \wedge \text{grant\_m}_5))
\end{array}
$$

Another style of *HOL_Theorem* is presented in Theorem 6. The theorem says that when AHB is initialised with the conditions described in *Assumption 1*, there will be exactly one master being granted bus ownership.

Theorem 6 (Mutual Exclusion)

$$
\begin{array}{l}
\textbf{AHB} \wedge \textit{Assumption(1)} \\
\rightarrow \\
GX(\bigvee_{0 \le i,j \le 5} grant\_m_i) \wedge GX(\bigwedge_{0 \le i,j \le 5, i \ne j} \neg(grant\_m_i \wedge grant\_m_j))
\end{array}
$$

For the remainder of Section 9.1, we use the notations employed in *Assumption 1* to describe the SMV fairness constraints and Theorem 6 to describe SMV theorems when they are imported into HOL.

9.1.2 Master Arbitration

After defining the initialisation process, the need to learn about the specific behaviour of the system will arise. The resources available 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 minimal information about the circuitry is available. One approach that can be taken is by performing experimental verifications using the documented specifications as guidelines. In this case, SMV is used to learn about the AHB system. When incorrect constraints are used in the verification, SMV generates a counter example. The documentation and feedback from SMV are used to determine the operational conditions of the system that can lead to the expected behaviours.

A master may request the arbiter to perform a burst process or a lock process. When the arbiter allows the master to perform these processes, the arbiter state machine goes into either a burst mode or a 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. The lock-split state is a condition when the arbiter is serving an active master lock request, with the slave responding 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 the slave issues a ready signal. This fairness constraint is shown in *Assumption 2*.

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

The transition of the arbiter's state machine into lock mode can be observed from the response to grant and lock signals of each master. When an active master is sending a lock signal, then the arbiter will go to lock mode. This condition is defined as ($\bigvee_{1 \leq i \leq 5}$ $(grant\_m_i \wedge lock\_m_i)$) and abbreviated as the *lock_req* signal. When *lock_req* goes to HIGH then the arbiter will be in 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 itself.

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 example, every time the active master de-asserts the lock signal, the slave responds with a split/retry signal. This condition forces the arbiter to go back into the lock or lock-split mode state. If this condition always occurs, the system will be trapped in lock mode. This condition was deliberately allowed to happen and the possible sequences needed to break this loop-trap were examined. The requirement to exit from the loop-trap is shown in *Assumption 3*.

Assumption 3.

$G\ (lock\_req \rightarrow F\ \neg lock\_req) \wedge$

$GF\ (lock\_mode \rightarrow (\neg grant\_m_0 \wedge slv\_ok/slv\_error \wedge \neg lock\_req \wedge$

$\quad X(slv\_ready \wedge slv\_ok/slv\_error \wedge \neg lock\_req)))$

First, when the active master asserts a lock signal, it will eventually de-assert itself. Second, the 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 the burst mode or able to exit from the lock-trap while in the lock mode. This condition is indicated by a *new_cycle* signal, when a HIGH output on this signal indicates that the arbiter is performing a new arbitration 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 shown below:

Theorem 7 (New Cycle)

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

There is a possibility that a granted master is forced by a slave into the split mode. When this condition occurs, the arbiter memorises which master has been split using the *split_m$_i$* signal. In this case, the arbiter will ignore all incoming requests from master m$_i$ until it receives an *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 where a master remains in split mode indefinitely, we define a new fairness condition which is shown in *Assumption 4*.

Assumption 4.

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

For every clock cycle, the arbiter evaluates the latest input signals and decides what action it will take. When a request sent by a master module is not granted, the module needs to keep requesting. This is because the arbiter does not memorise 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 shown in *Assumption 5.*

Assumption 5.

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

After the successful creation of the general scenario for a new arbitration, it can be used to obtain the requirements for the arbiter to grant every incoming master request. The additional rules are shown in Assumption 4 and 5. The general request-grant theorem is shown in Theorem 8. The theorem says that every master request will eventually be granted, provided all requirements defined in *Assumption 1* through *Assumption 5* are satisfied:

Theorem 8 (Masters' Request Liveness)

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

Theorem 8 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 9.3, a description will be provided in which Theorem 8 will be refined to construct an application specific verification platform.

The time taken to model check each of Theorem 7 and Theorem 8 was approximately 25 seconds.

9.1.3 Burst Transfer

In Theorem 7 and Theorem 8, we proved that under certain constraints the arbiter will always eventually perform a new arbitration and every request will always eventually be granted. These facts were used to verify the arbitration process for burst transfer. Based on the number of transfers, AMBA burst operations were categorised into four groups. The groups were: burst transfer of length one, burst transfer of length four, burst transfer of length eight, and burst transfer of length sixteen.

Because of the complexity of burst systems and the huge number of possible combinations of scenarios, it is extremly difficult to fully validate all possible combinations of burst transfers. One approach is to find the lower bound scenarios such as finding the minimum input sequences so that the burst transfer process can be completed. For example, a minimum input sequence to complete a four word burst transfer means the active master is granted access of the bus for at least four cycles.

Theorem 9 to Theorem 13 show the proofs of the highest arbitration priority master being granted access to the bus. In the first cycle, we use three initial assumptions. First, the arbiter is not being reset. Second, the arbiter is about to perform a new arbitration. Third, the expected active master is not in a split condition. Any split master can not be granted the bus until it is un-split. Provided these three conditions are satisfied, the request from master m_i will be granted in the next clock cycle. This property is shown in Theorem 9.

Theorem 9 (Burst Cycle 1)
$G((Reset \wedge req\_m_i \wedge \neg split\_m_i \wedge X\ new\_cycle) \rightarrow X\ grant\_m_i)$

The AMBA documentation [7] shows that when a master receives the grant signal at time t, the control signals are sampled at time (t+1). Unfortunately, the arbitration decisions were decided when the signals were sampled by the arbiter. The decision of whether the arbiter goes to burst mode has to be delayed until the next arbitration cycle. In addition to the first input sequence, the active master has to hold its request signal. Meanwhile, slaves are not allowed to respond with a split signal. In a burst transfer, a split signal will force the arbiter to terminate the transfer even if the transfer has not yet finished. The second burst cycle theorem is shown as follows:

Theorem 10 (Burst Cycle 2)
$G((Reset \wedge req\_m_i \wedge \neg split\_m_i \wedge X\ (new\_cycle \wedge req\_m_i \wedge \neg slv\_split)) \rightarrow X^2\ grant\_m_i)$

In the third burst cycle, the arbiter can previously be executing a burst of length one or going to the lock-mode. The arbiter is prevented from going to the lock-mode by assuming that the active master is not sending a lock signal and the arbiter is going to perform a new arbitration. In order to go to the burst-mode, the active master must send a transfer with a length of four or more and the transfer must be started with a non-sequential signal. The slave has to complete its previous transfer by sending a ready signal. Furthermore, the slave is not allowed to respond with a split signal. The theorem for the third burst cycle is shown in Theorem 11.

Theorem 11 (Burst Cycle 3)
$$\begin{aligned}
&G((Reset \wedge req\_m_i \wedge \neg split\_m_i \wedge \\
&\quad X\ (new\_cycle \wedge req\_m_i \wedge \neg slv\_split \wedge slv\_ready \wedge \neg lock\_m_i) \wedge \\
&\quad X^2\ (new\_cycle \wedge \neg slv\_split \wedge mst_i\_burst4+ \wedge mst_i\_nonseq \wedge \neg lock\_m_i)) \\
&\quad \rightarrow \\
&\quad X^3\ grant\_m_i)
\end{aligned}$$

In the fourth burst cycle, the arbiter should be in the burst-mode. The arbiter burst-mode behaviour only depends on the type of the master's transfer requests and the slave's transfer responses. The active master is only allowed in the state of busy or sequential and the active slave is only allowed to respond with okay or error. Otherwise, the arbiter will perform an early burst termination. The fourth burst cycle theorem is shown in Theorem 12.

Theorem 12 (Burst Cycle 4)
$$\begin{aligned}
&G((Reset \wedge req\_m_i \wedge \neg split\_m_i \wedge \\
&\quad X\ (new\_cycle \wedge req\_m_i \wedge \neg\ slv\_split \wedge slv\_ready \wedge \neg lock\_m_i) \wedge \\
&\quad X^2\ (new\_cycle \wedge \neg slv\_split \wedge mst_i\_burst4+ \wedge mst_i\_nonseq \wedge \neg lock\_m_i) \wedge \\
&\quad X^3\ (mst_i\_busy/seq \wedge slv_i\_okay/error)) \\
&\quad \rightarrow \\
&\quad X^4\ grant\_m_i)
\end{aligned}$$

The arbiter's behaviour for bursts of length eight and sixteen is similar to the burst of length four. The fifth to sixteenth burst cycle have similar behaviour to the fourth burst cycle. They only depend on the type of the master's transfer requests and the slave's transfer responses. The theorem for this group of burst cycles is shown in Theorem 13. The variable n used in the theorem is the cycle number. It ranges from 5 to 16.

Theorem 13 (Burst Cycle 4+)

$G((Reset \wedge req\_m_i \wedge \neg split\_m_i \wedge$

$\quad X\ (new\_cycle \wedge req\_m_i \wedge \neg\ slv\_split \wedge slv\_ready \wedge \neg lock\_m_i) \wedge$

$\quad X^2\ (new\_cycle \wedge \neg slv\_split \wedge mst_i\_burst8+ \wedge mst_i\_nonseq \wedge \neg lock\_m_i \wedge slv\_ready) \wedge$

$\quad \bigwedge_{4 \leq m \leq n} X^{m-1}\ (mst_i\_busy/seq \wedge slv_i\_okay/error))$

$\rightarrow$

$X^n\ grant\_m_i)$

The burst rule for the other masters is similar to the highest priority master. A lower priority master can be granted only when the higher priority master does not request to access the bus. The theorems for these masters are similar to the one described before. We only need to change (req_m$_i$) to (req_m$_i$ $\wedge \bigwedge_{i \leq j < max}$ req_m$_j$) where i is the master's priority number and max is the highest priority value. In RAPIER max has the value of 5.

80 burst theorems were proven, 16 theorems for each master. The proofs were completed after approximately 6 hours of CPU time.

9.1.4 Control and Data Transfer

The final group of properties that were checked is the correctness of the data transfer from the master to slave. Any master request which comes at time t can be granted at least one clock cycle after (time (t+1)). Then the control signal comes at time (t+2), followed by data at time (t+3).

In order to validate the transfer of control signals and data, it was assumed that no reset signal is applied to the arbiter and master m_i is granted the bus. The transfer process can start only when a previously granted master has completed its transfer. This can be determined by monitoring the ready signal from the active slave. In this stage the write, size and address are transfered from the master to the slave. The theorem for this proof is defined in Theorem 14

Theorem 14 (AMBA Control Signals)

$G((Reset \wedge grant\_m_i \wedge slv\_ready)$

$\quad \rightarrow$

$\quad X\ ((mst_i\_add = slv\_add)) \wedge (mst_i\_write = slv\_write) \wedge (mst_i\_size = slv\_size))$

Data output from the master to the slave and vice versa needs one additional cycle. A second ready signal is needed to show that the slave is ready to serve the request. The theorem for this proof is defined in Theorem 15.

Theorem 15 (AMBA Data Transfer)

$G((Reset \wedge grant\_m_i \wedge slv\_ready \wedge X\ slv\_ready)$

$\rightarrow$

$X^2\ ((mst_i\_wdata = slv\_wdata) \wedge (mst_i\_rdata = slv\_rdata)))$

The time used to model check Theorem 14 and Theorem 15 was approximately 60 seconds and 150 seconds respectively.

9.2 ARM7 properties

The ARM7 processor is the second core component of the verification platform. In AMBA AHB, the processor is defined as the default master and connected to the ports of m_1 through the ARM-AHB wrapper. The processor is modelled in ACL2 using the 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 ACL2 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 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, properties of the processor need to be obtained. They are obtained by proving facts using the ACL2 theorem prover. Three features of the processor that have been verified are presented below. All analyses were performed under the condition that no reset is applied to the processor. The processor's properties are as follows:

- The processor will continue its evaluation only when it receives 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. It was proven 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 the methodology, all components were combined and integrated in HOL. They were 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 the ACL2 model of ARM7, it needs to be transformed into a relational one.

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:

Definition 39 (HOL ARM7 processor module)

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

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 16 (ARM7 Properties)

$$\begin{aligned}
&\mathbf{ARM7} \wedge (G\ \neg reset) \\
&\rightarrow \\
&(G\ (\neg grant \rightarrow (Pst_{(t+1)} = Pst_t)) \wedge \\
&\ G\ (grant_{(t)} \rightarrow (\neg Plock(Pst_{(t+1)}) \vee \\
&\qquad (grant_{(t+1)} \rightarrow (\neg Plock(Pst_{(t+2)}) \vee \\
&\qquad\qquad (grant_{(t+2)} \rightarrow \neg Plock(Pst_{(t+3)}))))))
\end{aligned}$$

The processor is connected to the AMBA-AHB through its ARM-AHB wrapper. The communication between the processor and the bus protocol is moderated by the wrapper. Two properties which will be used in Section 9.3 are highlighted.

First, the wrapper is always sending a request signal to access the bus even if the processor does not need to access the bus. Second, the wrapper will never emit a busy signal to the AMBA-AHB. It always acknowledges the processor activity by only sending an idle, sequential, or non-sequential signal. The HOL theorem of these wrapper properties is defined in Theorem 17.

Theorem 17 (ARM-AHB_WRAPPER)
$G\ (\mathbf{WRAPPER} \rightarrow (req\_m_1 \wedge \neg mst_1\_busy))$

9.3 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 [13]. 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. The goal was to find the specifications or requirements for the memory controller and the slaves so that all desired properties were satisfied.

Interconnection of the components in the verification platform is a straightforward step. The formal models are connected and integrated with logical conjunctions in higher order logic. The integration of the AHB bus protocol, the ARM7 processor, and the ARM-AHB wrapper are just defined as ($\mathbf{AHB} \wedge \mathbf{ARM7} \wedge \mathbf{WRAPPER}$).

The goal was 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 the master's requests. These constraints were used 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. This constraint was used to refine existing AHB properties. The refinement is performed either using the model checker (SMV) or the theorem prover (HOL). In either case, existing properties were used and the constraints of the AHB bus protocol were simplified. There were no requirements to re-model check the bus protocol from scratch for the system with two masters. All proofs about AHB were imported into HOL where system level integration and verification were performed.

The non-existence of m_3 to m_5 meant that there was no request from any of these modules. One of the implications of this absence or simplification is that no grant signals were ever sent for these masters. The slave requirement of not allowing *split* or *retry* meant that a slave could only respond with *ok* or *error*. Because a slave is never emitting a *split* signal, no split condition will ever occur. In SMV it was proven that 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:

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

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 the 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, all signals related to the master can be ignored or turned off. The new assumption is shown below:

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

The restriction on slave modules of 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 the lock-mode depend only on the slave's ready signal and the master's lock request signal. The reduced fairness constraints are defined in *Assumption 7*.

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 depend only on the fairness condition of m_2 in not infinitely requesting the bus. Because m_1 is the default master, when m_2 does not request the bus, the arbiter will always grant the bus to the default master. The simplified assumption is shown in *Assumption 8*.

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

The behaviour of the wrapper is given by Theorem 17. One of the properties is that the wrapper 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.
$GF\ slv\_ready \wedge G(grant\_m_2 \rightarrow XF(\neg mst_2\_busy \wedge slv\_ready))$

The second wrapper property guarantees that the default master or processor is always requesting access to the bus. This property refines *Assumption 8* into *Assumption 10.*

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

Theorem 16 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 18. It states that the platform has two masters. It is constructed from the AHB bus protocol, the ARM7 processor, and the ARM-AHB WRAPPER. 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 18.

Theorem 18 (System's Properties)
AHB $\wedge$ **ARM7** $\wedge$ **WRAPPER** $\wedge$ *Assumption(6,9,10,11)*
$\rightarrow$
$G\ (F\ grant\_m_1 \wedge req\_m_2 \rightarrow F\ grant\_m_2)$

Based on Theorem 18, the requirements can be analysed and the specifications for each module can be defined. The second master (memory controller) has to satisfy the following specifications:

- The module has to be capable of 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 of 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 the following specifications:

- 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 slaves, an additional rule is defined which controls the behaviour of the slaves: when all inputs 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, the slave's output remains stable as long as the input does not change.

In this methodology, we obtain specialised specifications for both the master and slave modules. These 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 guarantees that the system fulfils the application's specific requirements.

9.4 Summary

In this chapter, the methodology was given to develop a generic formal verification platform in which applications can be developed. The generic platform behaviours were described as a set of formal properties. The generality of the properties makes 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.

A standard integration platform containing the AMBA-AHB bus protocol and an ARM7 processor was developed, with descriptions provided for the development of reusable formal properties for this platform. It was shown that the properties define the generic behaviour of the system. This platform was used to build an application and by evaluating the platform's properties with the application requirements, the specifications for the remaining components were obtained.

Chapter 10

Summary and Future Work

10.1 Summary

The primary objective of this dissertation is to demonstrate the feasibility of defining an integrated heterogenous formal tools system which can be used in the system level verification of SoC designs. The goal is achieved by addressing the problem in two steps. The first was to define a heterogenous formal system which can offer a complete set of formal technologies which are needed in the verification. The second was to use the heterogenous formal system in defining formal models and performing system level verification. In this dissertation, a tool architecture and methodology to perform formal verification for system on chip designs were presented.

A formal verification environment was defined. In this environment, various formal tools can be combined and integrated. It also allows each component to be modelled in the most suitable formalism. The formal verification environment used in this dissertation has the capability of symbolic simulation, model checking, and theorem proving. It combines the HOL98 theorem prover, the ACL2 theorem prover, and the SMV model checker. ACL2 is linked to HOL98 through the interface provided by PROSPER [27] and ACL2PII [76]. SMV is embedded in HOL as one of its decision procedures [70]. These interfaces provide an automatic mechanism for sharing information and reduce the possibility of errors being made during the translation of theorems from one formal tool to the other. The formal verification environment enables verification engineers to perform formal verification using the most suitable techniques at the component level and combine the results to achieve system level verification.

Two case studies, SIP and RAPIER, were formalised to demonstrate the application of a formal verification environment in system level verification. Both case studies were developed using the formal verification platform approach. In this approach, components were individually modelled in their most suitable formalism. A collection of formal models was developed as the building block for the formal verification platform. Two of these models are either for components most commonly used in all applications, such as processors and memory, or else are given in a generic way as properties of a component's external behaviours, such as a bus arbiter. The formal verification platform methodology provides a mechanism to wrap the models in such a way that they can be integrated and connected to create the verification platform. In this stage, system level properties of the combined system can be analysed.

Each case study was set to explore different aspects of system level verification for SoC designs. In the first case study (SIP), a complete system was formalised. The verification efforts were targeted to verify whether properties of the system as a whole satisfied the specifications. In the second case study (RAPIER), a partial system defined as a generic platform was formalised. Applications were developed by integrating additional components onto the platform. The formal verification platform for the partial system was used to obtain specific properties of the system. These properties can be used as guidelines for tighter specifications in the selection of components.

In SIP, a simple integration platform was developed by integrating two master modules, one slave module, an arbiter module, and a fragment of a software component embedded in the slave module. The verification showed that the system had liveness properties and that all master requests will eventually be granted by the arbiter. The program was also verified to ensure that it was correctly executed.

In RAPIER, a standard integration platform containing the AMBA-AHB bus protocol and an ARM7 processor was developed and descriptions were given of the development of reusable formal properties for this platform. The properties defined the generic behaviour of the system. This platform was used to build an application. By evaluating the platform's properties with the application requirements, the specification for the remaining components were obtained.

10.2 The methodology

The complexity of System on Chip design is arguably the biggest challenge in verification. The inclusion of embedded software in the design makes verification even harder. The methodology described in this dissertation differs from the one proposed by Liao and Hsiung [52]. They defined models as timed-automata models. The advantage of this modelling technique is that a multiple clock model and a gated clock model can be described. Verification is solely based on the use of a model checking tool. Similar to the work by Choi et.al. [19, 20], Liao and Hsiung did not address the issue of embedded software.

The methodology defined in this dissertation is based on a heterogenous formal verification environment which offers a complete set of formal verification technologies commonly used in hardware verification. The proposed formal verification environment combines three formal tools. The combined tools create a verification environment which has the capabilities to perform symbolic simulation, model checking, and theorem proving. The theorem prover enables the user to decompose a complex goal into more manageable sub-goals. The model checker can automate proving the sub-goals. The symbolic simulator can be used in validating the embedded software.

In this environment, components can be modelled in any of the supporting formalisms. Information on how the component will be verified is used to decide which formalism is the best suited to model the component. Subsequently, generic properties of each component are obtained. One approach is by defining the properties as input/output relations. The generality of the properties enables them to be used without the need to redo the verification whenever it is used to create a new design.

A formal verification platform for an SoC design is built by connecting formal models using higher order logic to compose relational predicates that model each component. When modelling with different formalisms is used, a layer of interface is needed to define the relationship between different formalisms. Using the verification environment, various modules described in various formal forms can be integrated and verified as a whole. The formal verification methodology allows the development of a generic formal verification platform in which applications can be developed. The generic platform behaviours were 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.

The formal verification approach has been applied successfully on SIP and RAPIER. SIP is a simple integration platform with system level complexity. The methodology is used to verify the protocol aspect of the system and the correct execution of a software application embedded in the system. The development of a scalable formal verification platform is presented in RAPIER. It uses models of an industrial standard microprocessor and bus protocol.

10.3 Future Work

The ARM verification platform is still in its early stages of development. A more comprehensive verification platform could be built on top of it. The platform will be based on the full specification of the AMBA bus protocol and the ARM7 processor. The platform could also be expanded by integrating standard peripherals. Embedded software will play an important role in shaping the generic platform to a specific application.

The formal verification platform methodology aims to create an environment where a verification platform can be built by integrating a selection of formal models in a 'plug and play' environment. A collection of reusable formal models and their formal proofs need to be developed to achieve the goal.

An obvious step in this work is to extend the system with more tools. The inclusion of more tools will enrich the system and provide more flexibility. There will be more choices of how to model the components and the techniques used in the verification.

One direction of interest is to develop a methodology for reusable embedded software systems. All embedded software implementations depend on the choice of the processor. The application code differs from one processor to another. A development framework of processor and software can increase software reuse.

Modelling a component, such as an ARM processor, is a difficult task. In most cases, the model usually fails to capture all behaviours of the component implementation, especially its specific implementation behaviours. Embedding the hardware description language, such as Verilog or VHDL, in the verification system allows the designer to obtain more realistic properties of the component. It will also foster the methodology closer towards the design flow.

Traditionally, interactive theorem proving systems only perform operations when a user sends an instruction. In most cases, the system is in an idle condition. The availability of fast processing computers should be exploited by letting the system perform background processes in trying to prove the remaining sub-goals. This will increase the automation of the theorem proving system.

The methodology has been used in building an ARM verification platform. Although the platform only uses the AMBA-AHB bus protocol specification, the methodology has shown its potential to be used in verifying a large design. A comprehensive study is needed to measure the scalability of the methodology and the system in use.

Bibliography

[1] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-John H. Seger. A Methodology for Large–Scale Hardware Verification. In Jr. Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design: Austin, Texas*, volume 1954 of *Lecture Notes in Computer Science*, pages 263–282. Springer–Verlag, November 2000.

[2] 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*, pages 538–541, June 1998.

[3] Mark D. Aagaard, Robert B. Jones, and Carl-John H. Seger. Lifted–FL: A Pragmatic Implementation of Combined Model Checking and Theorem. In Y.Bertot et al., editor, *Theorem Proving in Higher Order Logics: Nice, France*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340. Springer–Verlag, September 1999.

[4] ARM. *ARM–7 DataSheet*, DDI 0020C, December 1994.

[5] ARM. *Interfacing a Memory System to the ARM7TDMI without Using AMBA*, DAI 0029A, December 1995.

[6] ARM. *AHB Example AMBA System*, DDI 0170A, August 1999.

[7] ARM. *AMBA Specification ver 2.0*, IHI 0011A, May 1999.

[8] ARM. *AHB CPU Wrappers*, DDI 0169B, May 2001.

[9] Peter J. Ashenden. Modeling Digital Systems using VHDL. *IEEE Potentials magazine*, pages 27–30, April/May 1998.

[10] Janik Bergeron. *Writing Testbenches, Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[11] Mark Birnbaum and Howard Sachs. How VSIA Answers the SOC Dilemma. *IEEE Computer magazine*, pages 42–50, June 1999.

[12] Graham Birtwistle and Brian Graham. Verifying SECD in HOL. In Jorgen Staunstrup, editor, *Formal Methods for VLSI Design*, IFIP WG 10.5, Amsterdam, North–Holland, pages 129–177, June 1990.

[13] Richard Black. *System Integration Platform Pilot Technical Specification.* The Institute for System Level Integration, 1999.

[14] Bob Boyer and J Moore. Mechanized Formal Reasoning about Programs and Computing Machines. In R.Veroff, editor, *Automated Reasoning and its Applications: Essay in Honor of Larry Wos.* MIT Press, 1996.

[15] Muffy Calder and Alice Miller. Using SPIN to Analyse the FireWire Protocol - A Case Study. In *the IEEE 2394 FireWire workshop, Berlin*, pages 9–13, March 2001.

[16] Albert J. Camelleri. A Hybrid Approach to Verifying Liveness in a Symmetric Multi–Processor. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: Murray Hill, New Jersey*, volume 1275 of *Lecture Notes in Computer Science*, pages 49–67. Springer–Verlag, August 1997.

[17] 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.

[18] Pankaj Chauhan, Edmund M. Clarke, Yuan Lu, and Dong Wang. Verifying IP–Core based System–On–Chip Designs. In *the IEEE International ASIC/SOC Conference*, September 1999.

[19] Hoon Choi, Myung-Kyoon Yim, Jae-Young Lee, Byeong-Whee Yun, , and Yun-Tae Lee. Formal Verification of an Industrial System-on-a-Chip. In *the International Conference on Computer Design 2000, Austin, Texas*, pages 453–458, September 2000.

[20] Hoon Choi, Byeongwhee Yun, Yuntae Lee, and Hyunglae Roh. Model Checking of S3C2400X Industrial Embedded SOC Product. In *the 38th Design Automation Conference, Las Vegas, Nevada*, pages 611–616, June 2001.

[21] Windsor Chorlton. *The Invention of the Silicon Chip.* Heinemann, 2002.

[22] E. M. Clarke, O. Grumberg, H. Harashi, S. Jha, D. Long, K.L.McMillan, and L. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In Edmund M. Clarke, editor, *Formal Methods in System Design*, volume 06–02, pages 217–232. Kluwer, 1995.

[23] Edmund M. Clarke, O. Grumberg, and Doron A. Peled. *Model Checking.* The MIT Press, 2000.

[24] Edmund M. Clarke, D. E. Long, and K.L. McMillan. Compositional Model Checking. In *the 4th Annual Symposium on Logic in Computer Science: Pacific Grove, California*, pages 353–362, June 1989.

[25] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future. In *ACM Computing Survey*, volume 28–4, pages 626–643, December 1996.

[26] Ashish Darbari. Formaliziation and Execution of STE in HOL. In *Supplementary Proceeding Theorem Proving in Higher Order Logics: Rome, Italy*, September 2003.

[27] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems: Berlin, Germany*, volume 1785 of *Lecture Notes in Computer Science*, pages 78–92. Springer–Verlag, March/April 2000.

[28] David L. Dill. What's Between Simulation and Formal Verification? In *the 35th Design Automation Conference, San Francisco, California*, pages 328–329, June 1998.

[29] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *the IEEE International Conference on Computer Design: VLSI in Computers and processors*, pages 522–525, 1992.

[30] Anthony Fox. A HOL Specification of the ARM instruction Set Architecture. Technical Report 545, Computer Laboratory, University of Cambridge, United Kingdom, June 2001.

[31] Anthony Fox. An Algebraic Framework for Modelling and Verifying Microprocessors using HOL. Technical Report 512, Computer Laboratory, University of Cambridge, United Kingdom, March 2001.

[32] Anthony Fox. Formal Verification of the ARM6 Micro-architecture. Technical Report 548, Computer Laboratory, University of Cambridge, United Kingdom, November 2002.

[33] Steve Furber. *ARM System Architecture.* Addison–Wesley, 1999.

[34] M.J.C. Gordon and T.F.Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[35] Brian T. Graham. *The SECD Microprocessor; A Verification Case Study.* Kluwer, 1992.

[36] SoC Verification Business Unit Mentor Graphics. Design Challenges Thrust on SoC Process. *Nikkei Electronics Asia*, August 2000.

[37] David A. Greve. Symbolic Simulation of the JEM1 Microprocessor. In Ganesh Gopalakrishnan and Philip Windley, editors, *Formal Methods in Computer-Aided Design: PaloAlto, California*, volume 1522 of *Lecture Notes in Computer Science*, pages 321–333. Springer–Verlag, November 1998.

[38] Rajesh K. Gupta and Yervant Zorian. Introducing Core-Based System Design. *IEEE Design and Test of Computers magazine*, pages 15–25, October–December 1997.

[39] John Harrison. Formal Verification of Floating Point Trigonometric Functions. In Jr. Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design: Austin, Texas*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer–Verlag, November 2000.

[40] Scott Hazelhurst and Carl-Johan H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and OBDDs. Technical Report 93–41, Department of Computer Science, University of British Columbia, Canada, November 1993.

[41] IBM and Synopsys. *Design Environment for System–On–Chip.* White Paper on Synopsys Sucess Stories.

[42] The Institute for System Level Integration. *SLI Methodology Overview*, 2001.

[43] INTEL. *Expanding Moore's Law*, TL 001, 2002.

[44] C.Norris Ip and David L. Dill. Better Verification through Symmetry. In D. Agnew, Luc Claesen, and R. Compasano, editors, *Computer Hardware Description Languages and Their Applications, April 1993*, pages 87–100. Elsevier Science Publishers B.V., 1993.

[45] Robert B. Jones, John W. O'Leary, Carl-John H. Seger, Mark D. Aagaard, and Thomas F. Melham. Practical Formal Verification in Microprocessor Design. *IEEE Design and Test of Computers magazine*, pages 16–25, July–August 2001.

[46] Jeffrey John Joyce. *Multi–Level Verification of Microprocessor–Based Systems.* PhD thesis, University of Cambridge, 1990.

[47] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning, ACL2 Case Studies*. Kluwer, 2000.

[48] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning, An Approach*. Kluwer, 2000.

[49] Michael Keating and Pierre Bricaud. *Reuse Methodology manual For System–On–a–Chip Designs*. Kluwer, 1999.

[50] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *the ACM Symposium on Principles of Distributed Computing: Edmonton, Alta., Canada*, pages 239–247, August 1989.

[51] Pran Kurup, Taher Abbasi, and Ricky Bedi. *It's the Methodology, Stupid!* ByteK Designs, Inc., 1998.

[52] Wen-Shiu Liao and Pao-Ann Hsiung. FVP: A Formal verification Platform for SoC. In Ganesh Gopalakrishnan and Philip Windley, editors, *the 16th IEEE International SoC Conference: Portland, Oregon*, volume 1522 of *Lecture Notes in Computer Science*. Springer–Verlag, September 2003.

[53] Mark Litterick. *ARM Integration Platform Architectural Specification*. The Institute for System Level Integration, November 2001.

[54] Grant Martin, Lee Todd, and Andy McNelly. The Integration Platform Approach to System On Chip. In *IP98*, 1998.

[55] K.L. McMillan. Minimalist Proof Assistants: Interactions of Technology in Formal System Level Verification. In Ganesh Gopalakrishnan and Philip Windley, editors, *Formal Methods in Computer-Aided Design: PaloAlto, California*, volume 1522 of *Lecture Notes in Computer Science*, page 1. Springer–Verlag, November 1998.

[56] K.L. McMillan. *The SMV language*. Cadence Berkeley Labs, Cadence Design Systems, 1998.

[57] K.L. McMillan. *Getting started with SMV*. Cadence Berkeley Labs, Cadence Design Systems, 1999.

[58] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[59] Abdel Mokkadem, Ravi Hosabettu, and Ganesh Gopalakrishnan. Formalization and Proof of a Solution to the PCI 2.1 Bus Transaction Ordering Problem. In Ganesh Gopalakrishnan and Philip Windley, editors, *Formal Methods in Computer-Aided Design: PaloAlto, California*, volume

1522 of *Lecture Notes in Computer Science*, pages 237–254. Springer–Verlag, November 1998.

[60] J Strother Moore. Symbolic Simulation: An ACL2 Approach. In Ganesh Gopalakrishnan and Philip Windley, editors, *Formal Methods in Computer-Aided Design: PaloAlto, California, November 1998*, volume 1522 of *Lecture Notes in Computer Science*, pages 334–350. Springer–Verlag, November 1998.

[61] Vishnu A. Patankar, Alok Jain, and Randal E. Bryant. Formal Verification of an ARM processor. In *the 12th International Conference on VLSI Design, Goa, India*, January 1999.

[62] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[63] Douglas A. Pucknell and Kamran Eshraghian. *Basic VLSI Design.* Prentice Hall, 1994.

[64] 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 and Test of Computers magazine*, pages 26–35, October–December 1997.

[65] David Robinson. *Foundation Block and IP Library.* The Institute for System Level Integration, November 2001.

[66] Abhik Roychoudhury, Tulika Mitra, and S.R. Karri. Using Formal Techniques to Debug the AMBA System–on–Chip Bus Protocol. In *the Design, Automation, and Test Europe Conference, Munich, Germany*, March 2003.

[67] David M. Russinoff. A Case Study in Formal Verification of Register–Transfer Logic with ACL2: The Floating Point Adder of the AMD Athalon^{TM} Processor. In Jr. Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer Aided Design: Austin, Texas*, volume 1954 of *Lecture Notes in Computer Science*, pages 3–36. Springer–Verlag, November 2000.

[68] Jun Sawada. *Formal Verification of an Advanced Pipelined Machine.* PhD thesis, The University of Texas at Austin, December 1999.

[69] K. Schneider. Yet another look at LTL model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and verification Methods: Bad Herrenalb, Germany*, volume 1703 of *Lecture Notes in Computer Science*, pages 321–325. Springer–Verlag, September 1999.

[70] Klaus Schneider and Dirk W. Hoffmann. A HOL Conversion for Translating Linear Time Temporal Logic to ω–Automata. In Y. Bertot et al., editor, *Theorem Proving in Higher Order Logics: Nice, France*, volume 1690 of *Lecture Notes in Computer Science*, pages 255–272. Springer–Verlag, September 1999.

[71] Carl-Johan Seger. An Introduction to Formal Verification. Technical Report 92–1, Department of Computer Science, University of British Columbia, Canada, June 1992.

[72] 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. In Tiziana Margaria and Tom Melham, editors, *Correct Hardware Design and verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 534–537. Springer–Verlag, September 2001.

[73] Sonics Inc. *Open Core Protocol^{TM} Specification 1.0*, 1999.

[74] Madayam Srivas, Harald Rueβ, and David Cyrluk. Hardware Verification using PVS. In Thomas Kropf, editor, *Formal Hardware Verification Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 156–205. Springer–Verlag, July 1997.

[75] Mandayam K. Srivas and Steven P. Miller. Formal Verification of a commercial microprocessor. Technical Report SRI–CSL–95–04, SRI Computer Science Laboratory, July 1995.

[76] Mark Staples. Linking ACL2 and HOL. Technical Report 476, Computer Laboratory, University of Cambridge, United Kingdom, November 1999.

[77] Kong Woei Susanto. An Integrated Formal Approach for System on Chip. In *the IP Based Design 2002, Grenoble, France*, pages 119–123, October 2002.

[78] Kong Woei Susanto and Tom Melham. AMBA–ARM7 Formal Verification Platform. In *the 5th International Conference on Formal Engineering Method, Singapore*, November 2003.

[79] Texas Instrument. *TI's Standard–Independent Single Chip Digital Baseband Platform for Wireless System Design*, SPRY006 1996.

[80] University of Cambridge. *The HOL System Description, HOL98 Taupo–6*, June 2002.

[81] Alex van Someren and Carol Atack. *The ARM RISC Chip, A Programmer's Guide*. Addison–Wesley, 1994.

[82] Miroslav N. Velev, Randal E. Bryant, and Alok Jain. Efficient Modeling of Memory Arrays in Symbolic Simulation. In O. Grumberg, editor, *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 388–399. Springer–Verlag, June 1997.

[83] VLSI Technology. *Power and Flexibility in a Single Chip*, Product Bulletin PBGSM 1.0, October 1997.

[84] Jr. Warren A. Hunt. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[85] Jr. Warren A. Hunt and Bishop Brock. A Formal HDL and its use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, Prentice–Hall International Series in Computer Science, pages 35–48. Prentice–Hall, Englewood Cliffs, N.J., 1992.

[86] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP*. Addison–Wes ley Pub.Co., 1989.