Design of Data Abstraction Structure for MDG-HOL Hybrid Tool

SM Musabbir Hasan

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

September 2005

© SM Musabbir Hasan, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-20749-9
Our file Notre référence
ISBN: 978-0-494-20749-9

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

Design of Data Abstraction Structure for MDG-HOL Hybrid Tool

SM Musabbir Hasan

We have proposed design and implementation of a data abstraction structure that will result in extension to an existing Hybrid hardware verification tool so that it empowers to handle larger data paths automatically.

Interactive and user-expertise-dependent theorem proving techniques are well suited to handle large and complex data path dominated systems. However, they are complicated and difficult to handle when highly complex real-life designs are considered. On the other hand, automated state-space-exploration based techniques can verify trivial systems automatically, whereas they lack in the ability to verify practical designs due to state space explosion problems. To bring about a solution to the dilemma, hybrid approaches are under study, which widely vary in the tradeoff between the expressiveness of interactive approaches and automation and speed of the exploration based methodologies.

In the thesis we have described the design of an abstract data structure that allows natural numbers as the operands in addition to bit level descriptions. As a case study, we specified and implemented a generic computer processor using the abstract data structure. We implemented a parser that would be used to parse the specification and implementation of the design to be verified.

With the parser, and the data abstraction structure, it would be the perfect launch-pad for the implementation of a powerful and largely automatic tool that should be able to verify most practical hardware designs.

Acknowledgement

At the beginning, I must thank Dr. Abdeslam En-Nouaary for being my administrative supervisor. He forwarded invaluable suggestions targeting the perfection of the thesis. I would like to render my heartiest gratefulness to Dr. Skander Kort as he accepted me as his student and enabled a smooth entrance at Concordia University for me. Moreover, he guided me wisely and professionally through my courses so that I could complete my ground work and prepare well for the research. Finally he supervised and guided me throughout the whole research process, report writing and all other related logistic events.

Also, my gratitude is bound for my friends at the Hardware Verification Group in Concordia. They spent their own time and resources to extend cooperating hands in technical and logistic terms to me from time to time. Outside the group, I would like to acknowledge the cooperation of my friends who came forward with suggestions in both technical and non-technical aspects. Special thanks for my friends who proofread the report and suggested useful modifications in it.

Last but not the least; I won't be able to express how deeply I am indebted to my parents, my wife, and the rest of my family in Bangladesh. Their compassionate attitude, you-can-do-it approach, and we-are-there-for-you assurances drove me through unfavorable times and enabled me to hold the light of hope up during the odds.

I conclude this acknowledgement with heartfelt thanks to everyone I mentioned and apology to those I missed. Nonetheless, I am grateful to all the nice people around me who make my life livable.

Table of Contents

List of Figures		**ix**
List of Tables		**x**
Chapter 1	**Introduction**	**1**
1.1	Background and Motivation	1
1.2	Objective	2
1.3	Related Works	4
1.4	Scope of the Thesis	5
1.5	Organization of the Thesis	6
Chapter 2	**Theoretical Review**	**7**
2.1	Formal Methods	7
2.1.1	Synthesis	7
2.1.2	Formal Verification	10
2.1.2.1	General Description	10
2.1.2.2	Methodologies of Formal Verification	11
2.1.3	Theorem Proving	12
2.1.4	Model Checking	13
2.1.5	Symbolic Analysis	14
2.1.6	Binary Decision Diagrams	15
2.1.7	Partial Order Reduction	17
2.2	Abstraction in Model Checking	18
2.3	Decision Diagram-based Approaches	19

2.4	Theorem Proving vs. Model Checking	20
2.4.1	Theorem Proving	20
2.4.2	Model Checking	21
2.5	Hybrid Approach	22
2.6	Summary	23
Chapter 3	**Verification Methods**	**25**
3.1	The HOL Theorem Prover	25
3.2	Multi-way Decision Graph	29
3.3	Methodology	33
3.3.1	MDG Tactic Table	35
3.3.2	Data Abstraction Structure	37
3.4	Summary	38
Chapter 4	**Design and Implementation of Data Structure**	**39**
4.1	Overview	39
4.1.1	Sorts	42
4.1.2	MDG Word	42
4.1.3	MDG Value	43
4.1.4	MDG Function	43
4.1.5	Table Value	44
4.1.6	MDG Term	44
4.1.7	MDG Table	44
4.1.8	MDG Transform	45
4.1.9	Component	45

4.1.10	Block	45
4.2	Implementation of Proposed Data Structure	45
4.3	Issues Faced	46
4.4	Summary	47
Chapter 5	**Case Study**	**48**
5.1	Description of Simple RISC Processor	48
5.1.1	Control System	49
5.1.1.1	Control Signal Encoder	50
5.1.1.2	Clocking Logic	50
5.1.1.3	Control Step Generator	50
5.1.1.4	OpCode Decoder	51
5.1.2	Data Path	51
5.2	Formalization	52
5.2.1	Main Modules and Their Design	52
5.2.1.1	Instruction Set Architecture	53
5.2.2	Register Transfer Notation	55
5.2.3	Implementation	59
5.2.3.1	Controller	59
5.2.3.2	Data Path	60
5.3	Application Results and Analysis	60
5.3.1	Verification of an ALU	62
5.4	Summary	63
Chapter 6	**Conclusion and Future Work**	**64**

6.1 Major Contributions 64

6.2 Future Research Directions 65

Bibliography **67**

Appendix A **71**

Appendix B **81**

List of Figures

2.1	Pre-synthesis Model	7
2.2	Formal Synthesis Model	8
2.3	Formalization of VHDL	9
2.4.	Hierarchical Verification	23
3.1	Different Representations of an OR gate	31
3.2	Disintegration of Design into Blocks	35
3.3	Block Diagram of Arithmetic Logic Unit	35
4.1	Flow Chart of the Proposed Tool	40
4.2	Class Diagram for the Data Structure	46
5.1	Control Unit	49
5.2	Data Path	51
5.3	Instruction Formats	53

List of Tables

3.1 ALU OpCodes and Corresponding Function Symbols 36

3.2 Data Abstraction Structure 37

5.1 Register Transfer Notation 56

5.2 Truth table for control signal, PC_{out} 57

Chapter 1

Introduction

The introductory chapter of the report starts with stating the very background and motivation of the work. Then objective of the thesis and related works are discussed. Next we turn to the scope of the thesis followed by the way it has been organized focusing on the contents of each of the following chapters.

1.1 Background and Motivation

Hardware design is generally a large and complex procedure due mainly to the involvement of many different types of modules, intricate controls, exponential growth of chip sizes etc. When errors are introduced in a design at a late stage, it becomes catastrophically expensive to trace and fix, and the time-to-market is stretched in an unprecedented fashion.

Simulation is a conventional way of testing systems. However, when complexity of the design rises, coverage tends to be poorer and simulation time stretches beyond acceptability. Eventually bugs start to sneak in the design. "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give proof for its correctness"[1]. Moreover, it is not feasible to simulate all input sequences to completely verify a design. This is where the major roll of Formal Methods comes into play. Until now, it is a wide area of research because of the dispersal in direction, methodology and outcome offered by different techniques used for the purpose.

Among different formal methods, Theorem Proving and Model Checking techniques have gained more confidence and endeavor from the researchers. Theorem Proving starts with

building a mathematical model or theorem based on the structure and behavior of the system. A calculation on the derived theorem verifies if it satisfies certain desired properties. Theorem proving techniques can be automated to some degree, but it calls for in-depth knowledge and background of the user and it lacks in feedback in case of failure.

Model checking techniques are automatic. The systems being checked are assumed finite. Temporal logic is used for specifying the system properties. Owing to the automation feature, it does not require expertise of the user. However, it oversimplifies systems, which sometimes causes loss of significant detail. Model checking scales well for finite systems. For nontrivial systems, however, model checking may not be as useful due to state space explosion problem. A more efficient methodology evolves in the form of a hybrid approach that uses a series of abstraction last of which model checking can handle. Several research works have been directed towards finding an efficient methodology combining model checking and theorem proving approaches. However, among other drawbacks, data width problems have mostly deterred the success of the hybrid approach.

Thus, the prime motivation behind our work has been to design and implement a data abstraction structure that can eliminate the data width limitation problem of the hybrid tool. This would allow the tool to handle larger designs with automated methods and manage the overall proof with interactive approach.

1.2 Objective

Our main objective was to develop a data structure that enables the MDG (Multi-way Decision Graph) embedding in HOL with an aim to utilize the abstraction offered in it. The earlier version of embedding contained only concrete and abstract sorts in it. Concrete sorts, generally used for control signals, have explicit enumerations. Abstract sorts, on the other

end, are without any enumerations and are used for defining data sorts. To circumvent this limitation, our proposed structure stems on the formation of sorts that encompass, besides abstract and concrete ones, sorts that supports integers, words, Booleans and facilitates function declarations. Again, within the tool, the sorts are grouped into abstract and concrete depending on their intrinsic enumerations. To this end, Booleans, due to having a concretely defined enumeration, are interpreted as concrete sorts. Natural numbers are interpreted as abstract sorts. This allows us to deal with the operands, which are in natural number forms rather than as binary number of, for example, 32 bits.

The structure would allocate the choice to the user of the tool to define data at different abstraction levels according to the need of the proof procedure. Due to the fact that the data abstraction up to integers and refinement down to bits are defined within the data structure, verification scope is significantly enhanced. Depending on the outcome of a verification attempt with MDG the user might choose to either use a more abstract or refined model for the succeeding verification process. Thus, the automation of model checking is integrated with the tool whenever the abstract model is tractable.

Then we specified and implemented a generic simple RISC processor's single-bus 32 bit architecture in HOL incorporating the MDG embedding. The behavioral model is in terms of tables in MDG whereas the structural implementation is a net list of components. These components had also been defined as HOL definitions. Throughout the specification and implementation, the proposed data abstraction structure has been the imparted. These specification and implementation models offer to the designer of the tool an opportunity to apply on and gain in confidence plus conduct modifications as needed. Within the

implementation, we employed abstract function symbols. However, in a separate script, we provided interpretations, which would be deemed optional from the user's viewpoint.

1.3 Related Works

Some existing tools have successfully found bugs in implemented hardware systems. SMV [25], SPIN [26], COSPAN [27], HSIS [28], and VIS [29] are such proven tools that verified real-world examples [30, 31, 32] with more than 10^{30} states. Although the number seems enormous for control-oriented systems, this number of states is quickly exceeded if data paths are involved. In these cases, the verification with model checking tools often suffers from the state-explosion problem which roughly means that the number of states grows exponentially with the size of the implementation.

This work aims to extend the MDG-HOL hybrid tool, presented by S. Kort et al. [3], in which authors proposed an MDG-HOL hybrid tool that enables the user to verify a design in HOL hierarchically while using MDG to automate the proof of simpler sub modules down the hierarchy. In the paper, they demonstrated that with the application of the hybrid tool on a sample telecom switch block, verification was faster than the either cases when theorem-proving technique would be applied individually. [3]

T. Mhamdi presented a framework in which he attempted to balance the expressiveness of theorem proving and the efficiency and automation of state exploration techniques. He proposed to integrate a layer of checking algorithms based on MDG in HOL. He embedded the MDG underlying logic in HOL and implemented a platform that provides a set of algorithms allowing the user to develop own state-exploration based application inside HOL. While the verification problem is specified in HOL, the proof is derived by tightly combining

the MDG based computations and the theorem prover facilities. They have been able to implement different state exploration techniques within HOL such as MDG reachability analysis, equivalence and model checking. [21]

Y.Xu, X. Song, E. Cerny and O. Ait Mohamed studied model checking for a first-order linear-time temporal logic. They presented the computation model: ASM in which data and data operations were described using abstract sort and un-interpreted function symbols. They defined a first-order linear-time temporal logic called L_{mdg}, which supports the abstract data representations. Both safety and live ness properties can be expressed in L_{mdg}, however, only universal path quantification is possible. Fairness constraints can also be imposed. The property checking algorithms are based on implicit state enumeration of an ASM and implemented using MDG. [22]

1.4 Scope of the Thesis

This thesis can be treated as an extension to the MDG-HOL Hybrid Tool for hierarchical verification [3]. The hybrid tool was bound by a limitation in terms of data width it could deal with. We have proposed a data abstraction structure that eradicates this shortcoming. Now the user of the tool would not have to simplify the design and risk losing significant detail in the process. We utilized the abstraction facilities within MDG and extended it further to incorporate abstract sorts. We also specified and implemented a generic Simple RISC Computer controller and data path using the proposed data abstraction as part of our case study.

1.5 Organization of the Thesis

The rest of the report is arranged in the following manner:

Chapter 2 provides theoretical review of testing and verification. Chapter 3, is all about the HOL and MDG, where the in-depth analysis of these tools is done. Also other works in a related field are summarized there. Chapter 4 focuses on the design of the tool structure with embedded data abstraction. Chapter 5 highlights the design, specification and implementation chronology of the case study, which is about a Simple RISC processor implementation. Finally, chapter 6 encompasses the conclusion and future research directions.

In this chapter, the thesis topic has been introduced. The background of the thesis stems on the fact that traditional techniques to test and simulate might fall short in terms of effectiveness when it comes to large systems. This leads to verification methodologies where the data is represented in a more abstract fashion. The chapter wound up with the forecast of the succeeding chapters. In the next chapter, an elaborate discussion of the theoretical knowledge base is furnished.

Chapter 2

Theoretical Review

In this chapter, a far-reaching review of the theoretical concepts is made with a view to create the basis of understanding for our work. As described earlier, due to the lack of coverage offered by conventional approaches of testing systems, formal methods are evolving as widely practiced.

2.1 Formal Methods

Development of computer systems encompasses one major aspect – 'correctness'. Formal Methods can be defined as the application of logic to the development of 'correct' computer systems. [23] Formal methods are used in microprocessor design, cache coherency protocols, telecommunications protocols, rail and track signaling, security protocols, automotive companies etc.

Besides their usage in specification, design and implementation of software and hardware systems, formal methods are widely used in verification.

Following section elaborates the use of formal verification at different stages of synthesis:

2.1.1 Synthesis

Formal verification can be classified into following three main categories:

i. Pre-synthesis: In this case the Computer Aided Design (CAD) tool is formally

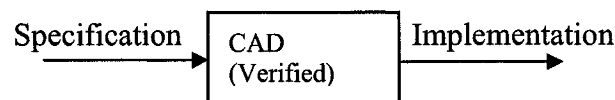


Fig 2.1 : Pre-synthesis Model

verified as shown in figure 2.1. Pre-synthesis has common use in software verification. In Pre-synthesis model, the verification process is guaranteed to succeed due to the fact that the CAD tool itself is verified.

However, some drawbacks of pre-synthesis are in the fact that CAD designers have to do everything and that it involves a large number of lines of codes.

ii. Post-synthesis: Formal hardware verification is grouped in this section. Here digital circuit designer verifies that the implementation implies the specification.

The major limitations of post-synthesis includes

a. The verification process must be repeated for each and every specifications
b. Post-synthesis proves difficult for complex circuits.

iii. Formal synthesis: This is accomplished in steps as shown in figure 2.2. In each step, resources are allocated and scheduling of the resources is completed. Then the steps are formalized. The final phase involves the proof of correctness of each step individually.

The main task entails proving that the output of each step is correct with respect to the corresponding input.

Fig 2.2 : Formal Synthesis Model

When an algorithm is fed as input, through high-level synthesis, one obtains an RTL (Register Transfer Level) description of the design. The effectiveness of formal synthesis is limited by the problem in definition and optimization of granularity of the steps. Optimizations can be Coarse-grained or Local (Fine) – grained.

Example of different levels of granularity:

$$S <= a * 4 \qquad (a)$$

$$S \leftarrow a \text{ lsl } 2 \qquad (b)$$

Expression (b) is the fine-grained optimized version of (a). In the expression (b), lsl stands for logical shift left.

Formal synthesis starts with a functional language. It must be mathematically defined. Conventional HDL (Hardware Description Language), such as VHDL (Very high speed integrated circuit HDL) can be a starting point too. However, VHDL is not a formal language, because its semantics are not mathematically defined- although some researchers tried to formalize it. Therefore, the formalization of VHDL progresses as shown in fig 2.3. VHDL is the informal description and a straightforward translation is desired from VHDL.

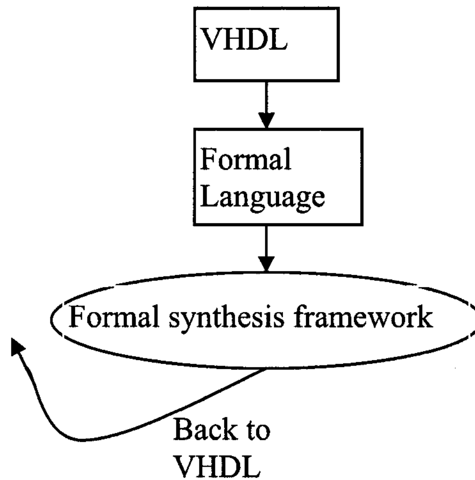


Fig 2.3 Formalization of VHDL

Therefore, the verification goal stands like:

Implementation => Specification

The expression above stands for, "the implementation does not contradict specification".

2.1.2 Formal Verification

Before going deep into verification methodologies, the general description of Formal Verification is elaborated below.

2.1.2.1 General Description

Formal verification is the process of checking whether a design satisfies some requirements (properties). Verifying the correct behavior of a digital system is becoming an open-ended task. Traditional techniques, such as simulation, are often inadequate for covering the large state spaces found in present day processor designs. Simulation and testing both involve making experiments before deploying the system in the field. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product. In the case of hardware verification, simulation is performed on the design of the circuit whereas testing is performed on the circuit itself. However, checking all of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible. Formal methods are emerging as a practical solution to specific aspects of the verification problem.

Formal verification is an approach to verify correct behavior in logic designs. Unlike simulation, where "confidence" comes from running an arbitrary number of test cases through a design, formal verification uses mathematical techniques to examine the entire solution space of a specified design property. There are no vectors. If formal verification says a property is verified, it is-under all conditions. Thus, while simulation is open-ended and uncertain, formal verification removes uncertainty, increasing designer confidence and reducing verification time.

2.1.2.2 Methodologies of Formal Verification

Among other methodologies, formal verification involves Interactive verification and automated techniques based on Binary Decision Diagrams (Described on page 15). Interactive verification usually refers to the use of axioms and proof rules to prove the correctness of systems. In early research on deductive verification, the focus was on guaranteeing the correctness of critical systems. It was assumed that the importance of their correct behavior was so great that the developer or a verification expert would spend whatever time was required for verifying the system. Initially such proofs were constructed manually. Eventually, researchers realized that software tools could be developed to enforce the correct use of axioms and proof rules. Such tools can also apply a systematic search to suggest various ways to progress from the current stage of the proof.

On the other hand, automated techniques include equivalence checking and model checking. BDD-based automated techniques are particularly suitable for verifying finite state concurrent systems. Example of such systems includes controllers with limited number of states. One benefit of this restriction is that verification can be performed automatically. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. Given sufficient resources, the procedure will always terminate with a yes/no answer. When the answer is 'No', the tool comes up with a counterexample which points to the flaw. Practically, a counterexample is a test vector with a sequence of states.

A third approach, sometimes called Hybrid approach, tends to combine interactive techniques and automated methods, which will be discussed in chapter 3.

2.1.3 Theorem Proving

Theorem proving technique progresses as a specification and its implementation are usually expressed as first-order or higher-order logic formulae. Their relationship, equivalence or implication, is regarded as a theorem to be proven within the proof system using a set of axioms and inference rules. Theorem proving has had its greatest successes in verifying data path dominated circuits as it supports the verification of parameterized data path dominated circuits. It can manipulate high abstraction and express powerful logic. Thus, theorem proving enables designs to be represented at different abstraction levels rather than only at the Boolean level. Moreover, it allows a hierarchical verification methodology, which can effectively deal with the overall functionality of designs having complex data paths. Theorem provers such as HOL (Higher Order Logic), Isabelle, PVS (Prototype Verification System) etc. are interactive techniques. However, in contrast to more automated formal verification methods, such as model checking or equivalence checking, it is currently a memory and time consuming method, especially when industrial designs are considered. Moreover, users need expertise to verify any design using theorem proving and this indicates to a limitation against applying the method on industrial designs.

Common approaches to Theorem Proving are-

- Fully Automated (Push-button technology)
- Human Assisted (Interactive Prover) - This mainly builds on step-by-step guidance. In this case systematic patterns of interactions can be captured as macros and it builds on tactics, strategies.
- Proof Checkers [4]

2.1.4 Model Checking

A model-checking problem is a problem of verifying that a formula f holds in a model M:

$$M \models f$$

where M represents the design and is usually finite, and f is the desired property of this design expressed through a temporal logic like CTL (Computation Tree Logic) or LTL (Linear Time Logic). [5]. Model checking techniques are based on the exhaustive state traversal of the model, and are often automatic and very efficient. The efficiency comes from the use of compact data structures like OBDDs (Ordered Binary Decision Diagrams) to represent the model, and powerful reduction techniques to reduce the search space. However, model checking is largely limited to finite models and propositional formulas.

Model checking as a verification technique has three fundamental features:

- It is automatic. It does not rely on complicated interaction with the user for incremental property proving. If a property does not hold, the model checker generates a counterexample trace automatically.
- The systems being checked are assumed to be finite. Typical examples of finite systems, for which model checking has successfully been applied, are digital sequential circuits and communication protocols.
- Temporal logic is used for specifying the system properties. [6]

Thus, model checking can be summarized as an algorithmic technique for checking temporal properties of finite systems.

Model Checking is an automatic technique for verifying finite state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic and the reactive system is modeled as a state transition graph.

An efficient search procedure is used to determine whether the state transition graph satisfies the specifications. The verifier provides a high level representation of the model and the specification to be checked. The model checker will either terminate with the answer true indicating that the model satisfies the specification, or, give a counterexample execution that shows why the formula is not satisfied. Two most important features distinct model checking technique from theorem proving ones-

- Model checking technique is highly automatic
- In case of non-compliance of the model under test with the specification, the checker provides counterexamples. Thus, it contributes to find the understated errors in complex reactive systems [7].

The major disadvantage of model checking relates its limitation to handle state space explosion problems.

2.1.5 Symbolic Analysis:

It has often been argued that model checking and theorem proving could be combined so that the former is applied to control-intensive properties while the latter is invoked on data-intensive properties. Obtaining a combination of theorem proving and model checking is not impossible. Both the techniques verify claims that look similar and it is possible to view model checking as a decision procedure for a well-defined fragment of specification logic. [24]. Nataranjan Shankar argued for a specific combination where theorem proving is used to reduce verification problems to finite state form and model checking explores properties of these reductions. This decomposition of the verification task forms the basis of the Symbolic Analysis Laboratory (SAL), a framework for combining different analysis tools for transition systems via a common intermediate language. Symbolic analysis is simply the computation

of fixed-point properties of programs through a combination of deductive and explorative techniques. The key elements of symbolic analysis are:

Automated deduction: computing property preserving abstractions and propagating the consequences of known properties.

Model checking: A means of verifying global properties of the system by means of systematic symbolic exploration. For this purpose, model checking is used for actually computing fixed points such as the reachable state set in addition to verifying given temporal properties.

Invariant generation: A technique for computing useful properties and propagating known properties. [2]

2.1.6 Binary Decision Diagrams (BDD)

A BDD is a data structure that is used to represent a Boolean function. The Boolean function is represented as a rooted, directed, acyclic graph in a BDD. It contains two types of vertices-terminal and non-terminal.

Canonicity reduces the semantic notion of equivalence to the syntactic notion of isomorphism.[12] If OBDDs (Ordered Binary Decision Diagram) are used as a canonical form for Boolean functions, then checking equivalence is reduced to checking isomorphism between binary decision diagrams. Similarly, satisfiability can be determined by checking equivalence to the trivial OBDD that consists of only one terminal labeled by 0. OBDDs are often much more compact than traditional normal forms such as conjunctive normal form and disjunctive normal form. They can be manipulated very efficiently. Therefore, they became widely used for a variety of CAD (Computer Aided Design) applications including symbolic simulation, verification of combinational logic and verification of sequential

circuits. An OBDD is similar to a binary decision tree except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf.

The size of an OBDD can depend critically on the variable ordering. Finding an optimal ordering for the variables is intricate. Several heuristics have been developed for finding an efficient variable ordering when such an ordering exists. If the Boolean function is given by a combinational circuit, then heuristics based on a depth-first traversal of the circuit diagram generally give good results. The intuition for these heuristics comes from the observation that OBDDs tend to be small when related variables are close together in the ordering. The variables appearing in a sub circuit are related in that they determine the sub circuit's output. Hence, these variables should usually be grouped together in the ordering. This may be accomplished by placing the variables in the order in which they are encountered during a depth-first traversal of the circuit diagram. A technique called dynamic reordering appears to be useful in those situations where no obvious ordering heuristics apply. When this technique is used, the OBDD package internally reorders the variables periodically in order to reduce the total number of vertices in use. The reordering method is designed to save time rather than to find an optimal ordering.

Reduced OBDD (ROBDD) is used in model checking to describe the transition and the output relation of a transition system. It can be viewed as a form of deterministic finite automata. An n-argument Boolean function can be identified with the set of strings in $\{0,1\}^n$ that evaluate to 1. Since this is a finite language and since all finite languages are regular, a minimal finite automaton accepts this set. This automaton provides a canonical representation for the original Boolean function. Logical operations on Boolean functions can

be implemented by set operations on the languages accepted by the finite automata. For example, AND corresponds to set intersection. Standard constructions from elementary automata theory can be used to compute these operations on languages. The standard OBDD operations can be viewed as analogs of these constructions. [8]

2.1.7 Partial Order Reduction

Model checking has proven to be almost perfect for the verification of small to medium scale systems. Nevertheless, with the increase in the complexity and rise in the number of states, one thing that undermines the effectiveness of Model checking tools is State Space Explosion. To prevent Model checking to be intimidated by state space explosion problem several measures are under trial- among them 'Partial order reduction' reduces the size of the state space that needs to be searched by model checking algorithms. It utilizes the commutative characteristics of concurrently executed transitions that result in the same state when executed in different orders. In synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved. Therefore, partial order reduction technique suits best for asynchronous systems.

The technique consists of constructing a reduced state graph. The full state graph may be too big for the memory, so it is never constructed. The behaviors of the reduced graph are a subset of the full graph. It can be shown that there exist behaviors, which are equivalent to each other. The checked property cannot distinguish between such behaviors. So for the checker to be correct, it is required that if a behavior is not present in a state graph, one equivalent of that must be included.

Partial order reduction is based on dependency relation that exists between the transitions of a system. In addition, the reduction method specifies which states would be included and which would not. [8]

2.2 Abstraction in Model Checking

Abstraction is the simplification of a system model by removing "irrelevant" detail. It reduces the state space while preserving essential characteristics of the system. Sometimes it is "obvious" that a detail can be removed. Abstraction is one of the most important techniques for reducing the state explosion problem. Two main techniques for abstraction are- The cone of influence reduction and data abstraction. Both of these techniques are performed on a high level description of the system, before the model for the system is constructed. Thus, the construction of the unreduced model that might be too big to fit into memory is avoided.

The cone of influence reduction attempts to decrease the size of the state transition graph by focusing on the variables of the system that are referred to in the specification. The reduction is obtained by eliminating variables that do not influence the variables in the specification. In this way, the checked properties are preserved, but the size of the model that needs to be verified is smaller.

Data abstraction, on the other hand, involves finding a mapping between the actual data values in the system and a small set of abstract data values. By extending this mapping to states and transitions, it is possible to obtain an abstract system that simulates the original system and is usually much smaller. Because of the reduction in size, it is frequently easier to verify the abstract system than the original system.

Lakhnech, Bensalem, Berezin and Owre put forward an incremental verification technique that uses abstraction of infinite-state and very large systems. It consists in finding an abstraction relation and an abstract system that simulates the concrete one and that is amenable to algorithmic verification. [13] One then checks that the abstract system satisfies the concrete property of interest. Well-established preservation results allow then to deduce, for a large class of properties, that the concrete system satisfies the concrete property if the abstract system satisfies the abstract one.

2.3 Decision Diagram-based Approaches

The MDG (Multi-way Decision Graphs) verification is a black-box approach. During the verification, the user does not need to understand the internal structure of the design being verified. The strength of MDG is its speed and ease of use. The MDG hardware verification system has been used in the verification of significant hardware examples such as Telecom System Block (TSB)—Receive APS Control, Synchronization Status Extraction and Bit Error Rate Monitor Telecom System Block (RASE TSB) which is a commercial product of PMC-Sierra Inc.[9]. A fundamental primitive of its hardware description language is the table which is the truth table representation of a relation between the values on variables. Used with don't-care and default values, next state variables and variable entries, it becomes a powerful specification construct that can be used to give behavioral specification of hardware as abstract state machines.

MDGs have been proposed as a solution to the data width problem of ROBDD based verification tools. The MDG tool combines the advantages of representing a circuit at higher abstraction levels as is possible in a theorem prover and of the automation offered by ROBDD based tools. An MDG is a finite, directed acyclic graph (DAG). MDGs essentially

represent relations rather than functions. They can also represent sets of states. They are much more compact than ROBDDs for designs containing a data path. Furthermore, sequential circuits can be verified independently of the width of the data path. The MDG tools combine the basic MDG operators and verification procedures. The verification procedures are combinational and sequential verification. The combinational verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. [10]. Further discussion on MDG is put in the third chapter.

2.4 Theorem Proving vs. Model Checking

This section is dedicated to the comparative analysis of theorem proving paradigm and model checking techniques.

2.4.1 Theorem Proving

It is particularly useful for architectural design and verification of the system. In case of theorem proving, both implementation description and specification description are developed through formal logic.

Correctness: $\vdash Imp \Rightarrow Spec$ (implication) or $\vdash Imp \Leftrightarrow Spec$ (equivalence)

It is possible to accommodate high abstraction level, expressive notation, powerful logic and reasoning. For the user to be able to manipulate Theorem Prover effectively, a deep understanding of design and logic is required as the tools are interactive. The user needs to develop rules tactics and lemmas for class of designs.

2.4.2 Model Checking

Model checking is specific to RT-level (or below) with at most ~400 Boolean state variables. Here the process implementation description is modeled as FSM whereas specification description is developed through properties in temporal logic.

Correctness: *Implementation* $\models$ *Specification* (property holds in the FSM model)

Model Checking is easy to learn and to apply as it is completely automatic. But properties must be carefully prepared for the model checking to be effective. It is integrated with design process and is in fact a refinement from skeletal model. The most important problem faced in model checking is state space explosion problem. It ceases to be useful when circuit grows large. However model checking helps increase confidence. [11]

In short, model checking is a powerful technique, but, it "oversimplifies". So it is recommended to try the best of both worlds approach. It works with a series of abstractions, the last of which model checkers can handle. Then theorem prover goes on to prove that each abstraction is property-preserving.

The weakness of Automated Theorem Provers is that it needs considerable proof effort, its proof finding process must be guided, and the user must be sophisticated. Moreover there is a lack of useful feedback in case of failure.

Model Checking is a powerful technique but it oversimplifies the problem. So, the best approach would be found in a useful and optimum combination of both the Model checking and theorem proving techniques. It can be done by a series of abstractions, the last of which model checkers can handle. And then prove (using theorem provers) that each abstraction is property preserving

2.5 Hybrid Approach

The discussions above illustrates clearly that both the automated decision diagram based formal hardware verification and interactive theorem proving methodologies have their own pros and cons.

Automated methods are fast and convenient, but have limitations for nontrivial systems, especially where data path and control circuitry are combined. Details of the version of the design under verification need to be simplified. Finding a model reduction and appropriate abstractions so that verification is tractable with the tool can be time consuming. Moreover, significant detail can be lost.

On the other hand, theorem prover can verify hierarchically allowing large designs to be verified without simplification. Furthermore, it is possible to reason about high-level abstractions of data types. It can however be very time-consuming, requiring significant user interaction and skill. Aiming to decimate these shortcomings, research works have been going on for a while around a hybrid approach such that the weaknesses of both approaches can be limited with the advantages being employed wherever applicable.

Several approaches to combine or hybrid the two formal verification techniques are in focus of recent research. Some, of these, concentrate on integration of model Checkers into theorem provers, while others work on integration of theorem proving techniques into model checkers. Nonetheless, the embedding of MDG into HOL, proposed by S. Kort et al. [3] to allow the combination of HOL's expressiveness and MDG's automation still remains distinctive from similar methodologies. This hybrid tool supports hierarchical verification within HOL while MDG is called to prove tractable modules and these verification results are integrated into overall HOL proof. They verified a communication block design using the

tool and demonstrated the improvement in speed and ease-of-use offered by it. Despite all its novelty, however, the tool suffered a drawback in terms of ability to handle data. It did not support data abstraction, which meant that all the data were assumed to have 1 bit instead of, say, 32 bits for the sake of simplification. However, chance of losing significant detail is more in that proposition.

One important aspect of the MDG-HOL hybrid tool was the hierarchical verification capacity. Generally, HOL is used to verify a design which is modeled as a hierarchy structure with modules divided into sub modules as shown in Fig 2.4

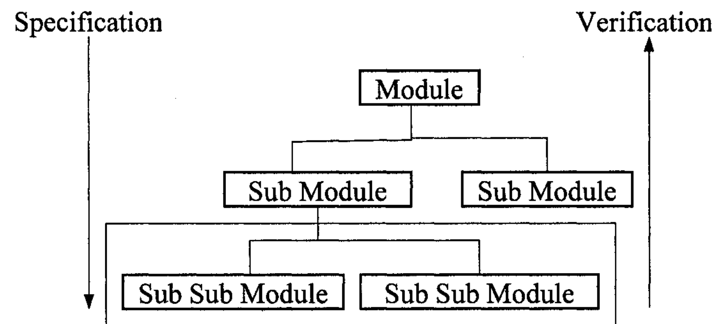


Fig. 2.4 Hierarchical Verification

The sub modules are repeatedly subdivided until eventually the logic gate level is reached. Both the structure and behavior specifications of each module are given as relations in higher-order logic. The verification of each module is carried out by proving a theorem that asserts that the implementation, its structure, implements (implies) the specification, its behavior. More discussion on hierarchical verification is furnished in chapter 3.

2.6 Summary

The main objective here was to prepare the basis of the proposed data abstraction aimed to be utilized in a verification tool. The procedural chronologies of a formal synthesis have been illustrated which formed the foundation of formal hardware verification. Relative review of

automated techniques and interactive approaches paved the path of a hybrid line of attack that aims to utilize the returns of both paradigms. From here, we can start delving deep into individual methods and gradually move towards our proposed methodology.

Chapter 3

Verification Methods

Previously, we have seen how the automated verification methodology differs from interactive ones. With the formal verification methodology in general in the context, this chapter focuses on the description of the two specific techniques that constitutes the existing MDG-HOL hybrid tool. With that picture in mind, the proposed methodology is introduced later in the chapter

3.1 The HOL Theorem Prover:

Theorem proving has had successes in verifying data path dominated circuits as it supports the verification of parameterized data path dominated circuits. It can manipulate high abstraction and express powerful logic. Thus, theorem proving enables designs to be represented at different abstraction levels rather than only at the boolean level. To this end, it allows a hierarchical verification methodology, which can effectively deal with the overall functionality of designs having complex data paths. However, in contrast to more automated formal verification methods, such as model checking or equivalence checking, it is still considered to be a memory-and-time-consuming method, especially when industrial designs are considered. Users need substantial expertise to verify any design using theorem proving and this fact points to a major limitation against applying the method on industrial designs.

Some of the widely used theorem provers in the hardware verification community are HOL (Higher-Order Logic), PVS (Prototype Verification System), Nqthm (a Boyer- Moore

theorem prover), ACL2 (Industrial strength version of the Boyer-Moore theorem prover) [20], HOL light (Intel) etc.

HOL can be precisely defined as 'an LCF (Logic for Computable Functions)-style proof for classical higher order built on top of (polymorphic) simply-typed λ-calculus'. [14]

HOL system is a general verification technique that accepts specification and implementation expressed as first-order or higher order logic formulae. The equivalence or implication between specification and implementation is regarded as a theorem to be proven within the proof system using a set of axioms and inference rules. Due to its generality, HOL is being used in a variety of application areas although it had been originally intended for hardware verification. HOL's interface to the system is the functional language ML. [15]

Robin Milner developed the approach to mechanizing formal proof used in HOL. He also headed the team that designed and implemented the language ML. That work centered on a system called LCF. LCF was intended for interactive automated reasoning about higher order recursively defined functions. With the purpose in view that other logics eventually be tried in place of the original logic, the interface of the logic to the meta-language was made explicit. The HOL system is a direct descendant of LCF; this is reflected in everything from its structure and outlook to its incorporation of ML, and even to parts of its implementation. Thus, HOL fits the preceding plan of application of LCF methodology to other logics. [16]

HOL supports both forward and goal-directed backward proofs in a natural-deduction-style calculus. In forward proof, the steps of a proof are implemented by applying inference rules chosen by the user and HOL checks the steps for safety. Derived inference rules are built on top of a small number of primitive inference rules. Backward proof procedures are carried out with user-defined tactics applied to proof goals or sub-goals. In the backward proof, the

user sets the desired theorem as a goal. Tactics are applied to the goal to create sub-goal and inference rules are applied to prove the sub-goals, which in turn proves the main goal. A tactic is usually a small SML program, which constitutes a high-level proof step. 'Tacticals' (it can be expressed as a sequential set of tactics) are repeatedly applied to break the goal into a list of sub-goals until they can be resolved. Theorem proving technique can be described as the procedure that involves constructing a mathematical model (M) of the computer program, hardware, or system concerned, and then using calculation to determine whether the model satisfies desired properties (P).

It constitutes of the following aspects:

•Model using "formal language" (logic)

•Calculation ($\vdash$) that takes up "formal deduction" (proof)

The prime issues of Theorem proving include expressive power (Propositional, First Order, Higher Order, Modal, etc.), decidability and complexity.

The end-result from a theorem prover must be sound (meaning 'what I prove is correct') and complete (meaning 'if something is correct, I can prove it'). A broader approach to modeling reveals that:

Background + Environment + System $\vdash$ Requirement

Fidelity of modeling requires background (domain knowledge, e.g. interpreted theories such as arithmetic and data types). Model checking over-simplifies the task and, in that way, downscales the complexity though it does fine for finite control. However, it is unable to handle data types and computations easily (extensions such as Binary Moment Diagrams needed to handle multiplier circuits). Finite models are another major constraint. [3]

Although, HOL can be used for directly proving theorems, but more often its role is as a theorem proving environment for implementing special purpose formal verification systems.

Types:

The types of the HOL logic are expressions that denote sets. There are four kinds of types in the HOL logic.

1. Type variables denote arbitrary sets in the universe. These are part of the meta-language and are used to range over object language types.

2. Atomic types stand for fixed sets in the universe. Each theory determines a particular collection of atomic types.

3. Compound types have the form $(\sigma_1,\ldots,\sigma_n)$ op, where $\sigma_1,\ldots\sigma_n$ are the argument types and op is a type operator of arity n. Type operators denote operations for constructing sets. The type $(\sigma_1,\ldots\sigma_n)$ op denotes the set resulting from applying the operation denoted by op to the sets denoted by $\sigma_{1,..}\sigma_n$. For example, list is a type operator with arity 1. It denotes the operation of forming all finite lists of elements from a given set. Another example is the type operator prod of arity 2 which denotes the Cartesian product operation. The type $(\sigma 1, \sigma 2)$ prod is written as $\sigma 1 \times \sigma 2$.

4. Function types: If $\sigma 1$ and $\sigma 2$ are types, then $\sigma 1 \rightarrow \sigma 2$ is the function type with domain $\sigma 1$ and range $\sigma 2$. It denotes the set of all (total) functions from the set denoted by its domain to the set denoted by its range.

Terms:

Terms are expressions that denote the elements of sets that are used to denote types. A term can be a variable, a constant, a function (combination of terms) or a lambda abstraction.

3.2 Multi-way Decision Graph (MDG)

The MDG system is a verification tool, based on decision diagrams and is primarily intended for hardware verification. Multi-way decision graphs are an extension of Reduced Ordered Binary Decision Diagram (ROBDD) approach [17]. MDGs were proposed [33] as a solution to the data width problem of ROBDD based verification tools.

The most important advantages of MDG over ROBDD are the incorporation of abstract sorts and un-interpreted function symbols. These allow the tool to verify large state spaces. The MDG tool combines some of the advantages of representing a circuit at higher abstract levels besides the automation offered by decision diagram based tools. [18]

An MDG is a finite, directed acyclic graph (DAG). MDGs essentially represent relations rather than functions. MDGs can also represent sets of states. They are much more compact than ROBDDs for designs containing a data path. Furthermore, sequential circuits can be verified independently of the width of the data path. The MDG tools combine the basic MDG operators and verification procedures. The verification procedures are combinational and sequential verification. The combinational verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. [19]. MDG tool is broadly a black-box approach. User isn't required to be aware of the internal structure of the system.

The input language for MDG system is MDG-Hardware Description Language (MDG-HDL). It supports structural descriptions, behavioral abstract state machine (ASM) descriptions and a mixture of both of these. A structural description usually resides at the Register Transfer Level (RTL) with a net list of components connected by signals. A

behavioral description is given by a tabular representation of the transition/output relation or truth table. The goal of verification tool is to prove that the structural implementation satisfies the behavioral specification.

The MDG tool provides four applications for combinational and sequential hardware verification – ASM state space exploration, safety property checking, equivalence checking and combinational verification. One notable point here is that, ASMs are not another kinds or state machine; rather they represent a way of describing state machines at a higher level of abstraction.

The MDG-HDL comes with a large library of basic components such as logic gates, multiplexers, registers, bus drivers etc. In the language, a circuit is described including the definition of signals, components and the circuit outputs. Declaration of a signal is accompanied by its sort. Input and output ports of predefined components are instantiated for declaring components. Although there are some requirements on the node ordering for abstract variables, there is none for concrete variables. Abstract data types (ADT) are used to denote data values. Un-interpreted function symbols denote data operations. Cross-operators are used to represent feedback from the data path to control circuitry. They are un-interpreted; however, their allowable range of interpretation is limited by information provided by axioms.

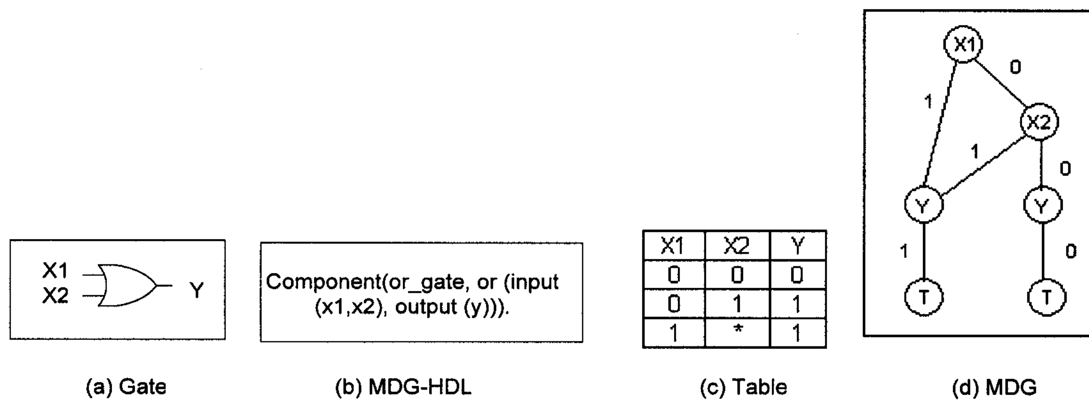
Sorts are generally either concrete or abstract. Concrete sorts are characterized by their enumerations. Abstract sorts do not have any enumerations.

'Table' is an important one among the predefined component modules. Both implementation and specification of a design can have tables to describe a functional block. A table is similar to a truth table. However, it allows first-order terms in its rows. A table contains a list of

rows. The first row consists of a list of variables and cross-terms. The last element of the list must be a (either concrete or abstract) variable whereas all the other variables must be concrete. Starting from the second row are lists of values that the corresponding variables or cross-terms can take. For example, following is a table to represent a 2-input OR gate:

table ([[x1,x2,y], [0,0,0],[0,1,1] [1,*,1])

The table description is further internally translated into an MDG (decision diagram) with the variable ordering x2<x2<y (figure 3.1 d)



X1	X2	Y
0	0	0
0	1	1
1	*	1

(a) Gate (b) MDG-HDL (c) Table (d) MDG

Fig 3.1 : Different Representations of an OR Gate

Another example of the use of tables with abstract variables and functions is given by the following example:

table ([[cnd,geq(x,y),n_y],[1,1,x]|y])

which defines the function

if (cnd = 1) and (geq(x,y) =1) then n_y = x else n_y = y.

where cnd is a concrete boolean variable, x is an abstract input variable, y is an abstract state variable, and n_y represents its next state. geq represents a function symbol that means

"greater-or-equal". The term y after symbol '|' in the table description is used as the default value.

MDG tool requires, beside circuit descriptions, a variety of information, such as sort and function type definitions, symbol ordering and invariant specifications, etc. These are provided in order to use all the applications outlined earlier. Supplied information are organized in six files: Algebraic specification file, Symbol order file, Circuit description file, Invariant specification file, State encoding file and Manual relation partitioning file. [20].

The algebraic specification file defines sorts, function types and generic constants used in hardware descriptions. It also includes, if necessary, the rewrite rules which partially interpret otherwise un-interpreted function symbols.

The symbol order file provides the custom symbol order for all the variables and cross-operators which would appear in MDGs.

The circuit description file declares signals and their sort assignments, component network, outputs and the mapping between state variables and next state variables. *table* is used as the tabular representation for behavioral descriptions.

The invariant specification file defines the invariant to be checked during the reachability analysis.

The state encoding file provides the mapping of state encoding between two circuits to be compared. The manual relation partitioning file allows users to declare partitioned transition/output relations manually for the product machine.

When MDG tool is used for the verification of a hardware design, it returns a counterexample in case the verification fails. A counter example service generates a

sequence of input-state pairs leading from the initial state to the incorrect behavior. This is handy when it comes to pinpointing the design flaws.

Non termination problem: It is an important issue as per as MDG tool is concerned. The abstract state enumeration method may not terminate. For example [20], if the program counter of a microprocessor is represented by a variable *pc* of abstract type, the initial value of *pc* is given by a generic constant *zero*, and a function symbol *inc* denotes the incrementing of *pc* by a non-branching instruction, then a node labeled by *pc* will have, in recursive invocations of the algorithms, an unbounded collection of edges *pc, inc(pc), inc(inc(pc)), etc.,* and the algorithm will not terminate. This can be avoided by generalizing the initial state, using a variable x to denote the initial value of the program counter. Generalizing the initial state amounts to supplying an invariant. In general case, however, there is no guarantee that an invariant can be found to avoid non-termination.

MDG size: Another important point to consider about the MDG, is the size of it. The efficiency and effectiveness of the verification process largely depends on the size of the MDG resulting from the transformation of the design specification. Several ways for reducing the size of the graph are contemplated, such as, symbol ordering, partitioned transition relations, ROM declaration, rewrite rules for cross-terms etc.

3.3 Our Methodology

Numerous research works in the area of formal hardware verification have been and are directed towards finding optimum balance between the mutually exclusive advantages of two paradigms in- user-interactive but able-to-handle-large-designs theorem proving techniques and largely-automated but state-explosion-prone model checking procedures. Furthermore,

works, intending to hybrid HOL with MDG in order to accumulate their individual advantages into one tool, have also been developed for some time now. In those researches, attention has been focused on embedding MDG into HOL, improving the temporal logic to make MDG more effective and seamless inside HOL etc. However, one important aspect of MDG's shortcomings is its limited ability to handle large data paths. Use of abstraction at various stages and aspects of the existing verification systems is a key option that has been researched for quite some time now. To this end, an efficient way of data abstraction, that enables the MDG tool to verify larger system, should be a remarkable solution.

The main goal of the verification procedure will be to prove the global goal for a given design that:

$$\vdash \text{Implementation} \supset \text{Specification} \qquad (1)$$

Rather than attempting to verify the whole design altogether, the tool would use the hierarchical verification tactic to recursively break down the top-level design into sub blocks until a tractable (to the proof tool) unit is attained.

Let us consider a design that is broken into sub modules Block 1 and Block 2 as shown in figure 3.1. While the main goal of the proof is same as expression (1), the HIER_VERIF_TAC [3] would derive following sub goals from the main goal:

I. $\vdash \text{Imp } 1 \supset \text{Spec } 1$

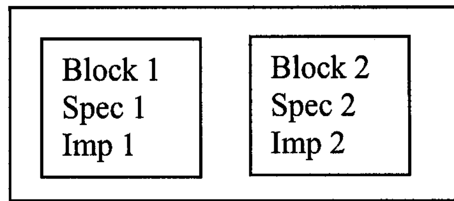


Fig 3.2 : Disintegration of a Design into Blocks

II. $\vdash$ Imp 2 $\supset$ Spec 2

III. $\vdash$ Spec1 $\wedge$ Spec 2 $\supset$ Specification

Since expressions I and II are implications, proving III would suffice to prove the top goal.

3.3.1 MDG Tactic Table

MDG tactic tables represent un-interpreted function symbols and their corresponding

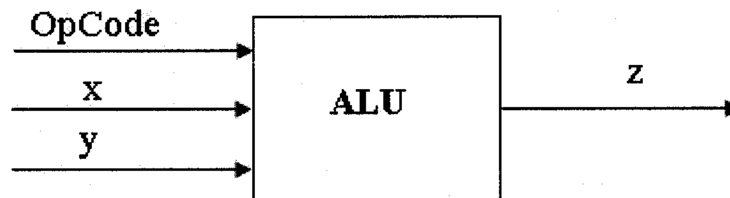


Fig 3.3 : Block Diagram of an Arithmetic Logic Unit

symbols in MDG tool. Let us consider a table for arithmetic logic unit (ALU) whose operands are x and y and the output is z. We assume that value of "OpCode" indicates the desired arithmetic operation.

Table 3.1 shows the MDG tactic table for the ALU (Fig 3.3) where the value of the output is given for different values of the OpCode.

OpCode	Z
add	add_fn (x,y)
sub	sub_fn (x,y)
mul	mul_fn (x,y)
div	div_fn (x,y)

Table 3.1 : ALU OpCodes and Corresponding Function Symbols

The sort of 'OpCode' must be concrete with the symbols appearing in the column as its enumeration, as expressed below:

conc_sort (OpCode, [add,sub,mul,div]).

In the table the symbols add_fn (x,y) is an un-interpreted function symbol that might mean addition. Also x, y and z must be of abstract data type, namely, wordn.

During verification, functions symbols are interpreted at different abstraction levels as required by the tools. For example:

add_n: It is interpreted as HOL add on 'num'. At this abstraction level, the proof requirement is

$\forall$interpretation. imp => spec

and when proved, a theorem (e.g. Ctrl_thm) is generated.

add_w : It is interpreted as HOL add on 'word's. Integers are refined to words by HOL operator NBWORD and words are abstracted to integers by BNVAL as required. For example integer '14' is translated to Boolean word [1,1,1,0]. At this stage the proof requirement looks like

$\text{Interpretation2}_{\text{Behavioral}} \Rightarrow \text{Interpretation1}_{\text{Behavioral}}$

and the proof should use the Ctrl_thm generated at the integer level.

add_i : It is interpreted as bit addition. At this level, operations are disintegrated further to obtain bit-level proofs. For example: in case of a 16-bit addition, the structure is re-arranged with four Carry Look Ahead (CLA_4) adders. CLA_4s are verified using MDG. Consequently, for each individual adder, it is proved that

$CLA_{4\text{-Structural}} \Rightarrow CLA_{4\text{-Behavioral}}$

3.3.2 Data Abstraction Structure

Embedded HDL	**HOL**	**HDL**
BOOL	Bool	Bit
NUM	Num	bus (num) [Not synthesizable]
BUS	Bit word	bus (wordn) [Defined bus width]
ABSTRACT	'a set	Abstract [Not synthesizable]
CONCRETE	Enumeration	Concrete

Table 3.2 : Data Abstraction Structure

The table demonstrates the formation of data sorts through different platforms of specification and implementation. Notable here are HOL numbers which would be translated as 'bus' of 'wordn', an abstract sort in MDG. Thus, natural numbers are incorporated within the definition. Again, a Boolean word in HOL that has a small length would be interpreted as

bus of concrete sort, whereas a larger word will be interpreted as abstract. However, abstract sorts are not synthesizable in the HDL.

3.4 Summary

This chapter sheds light on HOL and MDG tools. Then the proposed methodology for the design of data abstraction structure for MDG-HOL hybrid tool is introduced to show gradual transformation of data from hardware description language in general to the MDG-HDL that was used in the data structure and incorporated by the embedding. Now that we have discussed our methodology in general, coming chapters put focus on the design and implementation procedure followed by the issues faced during the procedure and how these were overcome.

Chapter 4

Design and Implementation of Proposed Data Structure

In the previous chapter, the focus was on HOL and MDG and based upon that description we went on to explain our proposed methodology. Now, we will continue into our proposed data structure in terms of design and implementation.

4.1 Overview

As described earlier, the tool links HOL and MDG with the aim in mind that the individual pros of each tool can be incorporated in it and be exploited to the optimum. The tool is written using Moscow ML. The overall proof procedure is managed within HOL whereas MDG is summoned seamlessly to prove relatively tractable components, sub blocks or blocks. The tool counts on ML functions, called tactics, to conduct the proofs. The specification and implementation of the design to be verified are written in HOL. MDG has been embedded in HOL in order to be able to support the semantics of HOL.

The design hierarchy, of the sub blocks that constitute a block implementation, dictates the way in which a proof is structured in HOL. Each and every block of the design is specified in terms of its behavior and structure. Blocks residing at the bottom of the hierarchy are considered primitive components. All the other block's implementations are verified against their specification in following three steps:

1. An intermediate verification result is obtained about the block based on the behavioral descriptions of its sub blocks considering sub blocks as primitive components.

2. The same process is repeated recursively on the sub blocks to obtain correctness theorems for them.

3. The correctness theorems of the sub blocks are combined with the intermediate correctness theorem of the block itself to give the actual correctness theorem of the block. This is based on the full structural description of the block down to primitive components.

The verification follows the natural design hierarchy as shown in figure 4.1. If this process is applied to the top-level design block, a correctness theorem for the whole design is obtained.

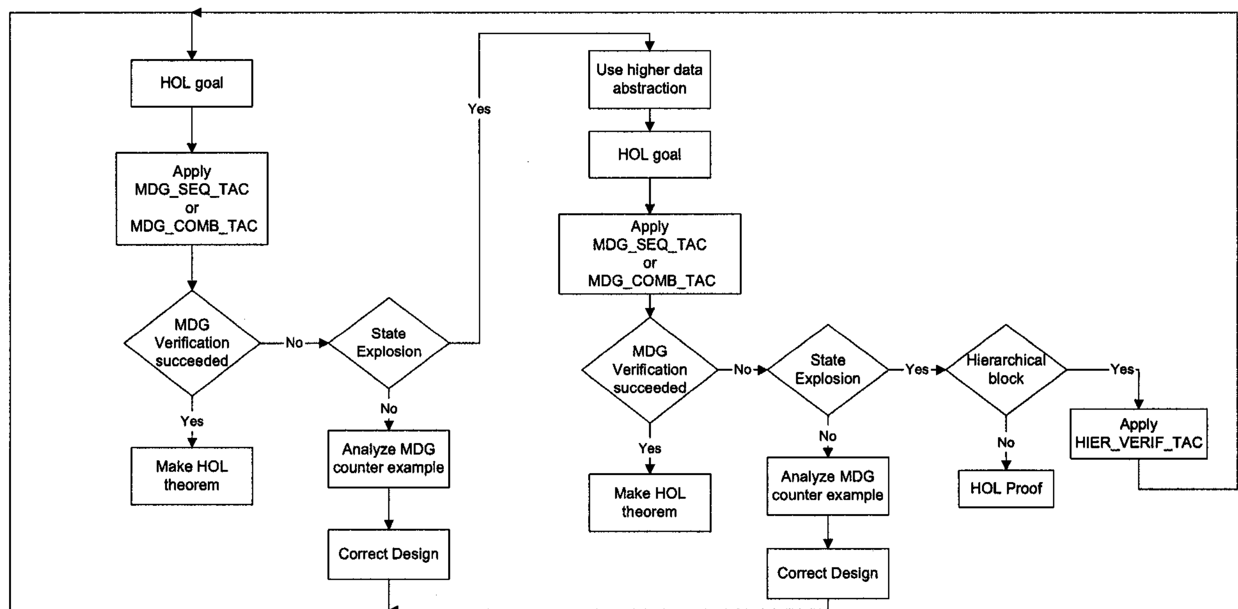


Figure 4.1: A Flow Chart of the Proposed Tool

The integration of the verification results of the separate components that would be done informally, if at all, in an MDG verification is thus formalized and machine-checked in the HOL approach.

The hybrid tool fits the use of MDG verification naturally within the HOL framework of compositional hierarchical verification [3]. Thus, it supports hierarchical verification

automating the process discussed earlier. There is no need to provide behavioral specifications for sub blocks and to verify them separately because the HOL system manages the proof with the MDG system called to verify tractable design blocks. At any point, hierarchical verification can be abandoned and the whole block can be verified in MDG provided the block is sufficiently simple. Besides, a tractable block can be verified using MDG under the assumption that its sub blocks are all primitive components. Since no information is lost in using MDG via the hybrid tool, a normal HOL proof can still be performed in case the block is not simple enough. To allow the seamless integration of the tools, we use MDG-style behavioral specifications within HOL. This warrants that the specifications must be in the form of a finite state machine or table description [3]. When state explosion occurs due to the application of MDG to a block, the user has the choice to either apply HIER_VERIF_TAC to break up in tractable modules or to use the same block with higher data abstraction applied to it.

The hybrid tool supports the hierarchical verification process by providing a HOL embedding of the concrete subset of the MDG input language to allow MDG-style specifications to be written in HOL. Three high-level proof tactics that manage the proof process are also provided. A hierarchy tactic HIER_VERIF_TAC automates the sub goaling of the correctness theorem of a block by analyzing its structure [3]. It later combines the proven sub goals to give the desired correctness theorem. Where a non-primitive component occurs several times within a block, the tactic avoids duplication generating a single sub goal that once proved is automatically instantiated for each occurrence of that component to prove the correctness of the block. Two other tactics automate the link to the MDG tools. MDG_COMB_TAC attempts to verify a given correctness theorem for a block using MDG

combinational equivalence verification. MDG_SEQ_TAC [3] calls MDG sequential equivalence verification to prove the result.

4.1.1 Sorts

This structure of Sort is of central importance to the tool's operation. In contrast to previous embeddings where sorts have been either abstract or concrete, here we define Boolean, integer, bus and product sorts in addition to abstract and concrete sorts. Concrete sorts may be user-defined sorts with explicit enumeration. Abstract sort enables abstraction of the data in order to provide capacity to handle larger data width. Bus sort defines the width of 'wordn' defined in MDG-HDL. Product sort facilitates the declaration of functions where the domain and range of the function have individual sort assignments. Integer is another important abstract sort, which enables the user to define inputs as integers so that verification in MDG can take place, at a higher level of abstraction, on functions like addition between two integers. Boolean sort, which was earlier included in concrete sort, is explicitly defined now. Other sorts can also be declared concrete provided they have explicit enumerations of their own. For example, Boolean sorts have [0,1] as their possible values. So, Boolean sort is grouped to be a concrete sort. Similarly, words of length up to 8, have their enumerations explicit in MDG. So, those are declared concrete whereas, larger words hold abstract sort. Integers are always dealt with as abstract sorts.

4.1.2 MDG Word

A new data type, 'MDG word' is declared that enables to handle HOL words. As HOL words are composed of a list of Boolean values, our MDG word constructs out of a list of characters. Each individual character member of the list corresponds to the Boolean value

declared in the HOL specification of the design. Method 'ToMDG' transforms a word into its equivalent integer. Thus, the user is offered the freedom of defining data in integers, words or bits.

4.1.3 MDG Value

An MDG value can be constructed from Boolean, integer, string, MDG words, abstract sorts, concrete sorts or recursive product of MDG values. In order to check the concreteness of a value, we wrote a function 'OfConcreteSort' which takes the value and the sort of it. If the value is concrete and the sort is concrete too and also the value is found within the enumeration of the sort, the function returns true; otherwise it returns false.

MDG constants are constructed from an MDG value along with its sort. A constant can be concrete or abstract depending upon its sort. Again, the constant- whose MDG value is concrete and the sort is concrete too- we call it an individual constant. The constant that is not an individual constant is defined as a generic constant.

4.1.4 MDG Function

An MDG function is declared using its name, sort of its domain and sort of its range. A function is defined to be a concrete function if all of its arguments are concrete and also it ranges over an argument of concrete sort. An abstract function must be ranged over an abstract sort. Cross-operators are allowed to have at least one abstract argument in its domain whereas the range must be concrete. Concrete function symbols must have explicit definition, while abstract function symbols and cross-operators are un-interpreted.

4.1.5 Table Value

Table values are MDG values used in tables. A table value must be either an MDG value or a 'Don't_care' value. 'Don't_care' specifies the situations when the value associated to a particular input in a table doesn't influence the output.

4.1.6 MDG Term

An MDG term can be a constant, a variable, a function or a recursive product of MDG terms. In order to check the well-typed-ness of a function term, the method 'WellTypedTerm' ensures that the term is of the same sort as the range of the function and the term is a well-typed one.

4.1.7 MDG Table

A table consists of five arguments namely, input list, output, input rows, output column and default value. The input list is a list of terms. Casting functions are required to convert variables into MDG Terms. We defined these casting functions in a separate module. The output is a MDG variable. Input rows are of list of 'Row's. Consequently, a row is a list of Table values. In the case study, we utilized the casting of different parameters into Table values using the constructor 'TABLE_VAL'. The output values corresponding to the input combinations are given as a list of terms. Finally, the default value is again a term.

Individual methods are defined so each of the five arguments can be extracted from a table when necessary. The function, 'ToMDG', converts a table definition to a string that respects the syntax of table construct in MDG.

4.1.8 MDG Transform

An MDG transform is a black-box component that implements any function. An MDG transform is constituted by a function, its inputs and the output.

4.1.9 Component

A component can be an MDG component, a multiplexer, a table or a function expressed in transformation form. It is possible to retrieve the signals and their corresponding sorts from a component with respect to what type of component it is. For example, if the component is an MDG component and its function is to return the OR-ed output for two words, the sort must be word too. Again, for a table that contains only control signals, the signals are associated to Boolean sorts.

4.1.10 Block

The proposed tool extracts block from the design when it is parsed by the parser. Blocks are characterized by an interface, group of sub blocks, internal signals and sorts of the signals.

Interface: The interface of a block consists a name, the input and output ports for the block.

Sub block: A sub block can be another block or a component.

Internal Signals: The sub blocks are connected by internal signals.

Sorts of Internal Signals: This field stores the sort information for the block. When a component is extracted as a block, depending of the component characteristics, the sorts of its inputs and outputs are determined.

4.2 Implementation of Proposed Data Structure

Figure 4.2 shows a class diagram for the proposed data structure. It is shown that Block is formed by sub blocks, Block interface has name and input and output ports, and component can be a multiplexer, an MDG component, a Table or a function represented as a black box.

An MDG Table uses variables, terms and table values for its construction. An MDG constant is formed with an MDG value along with its associated sort. Furthermore, Table Values are constructed with MDG values.

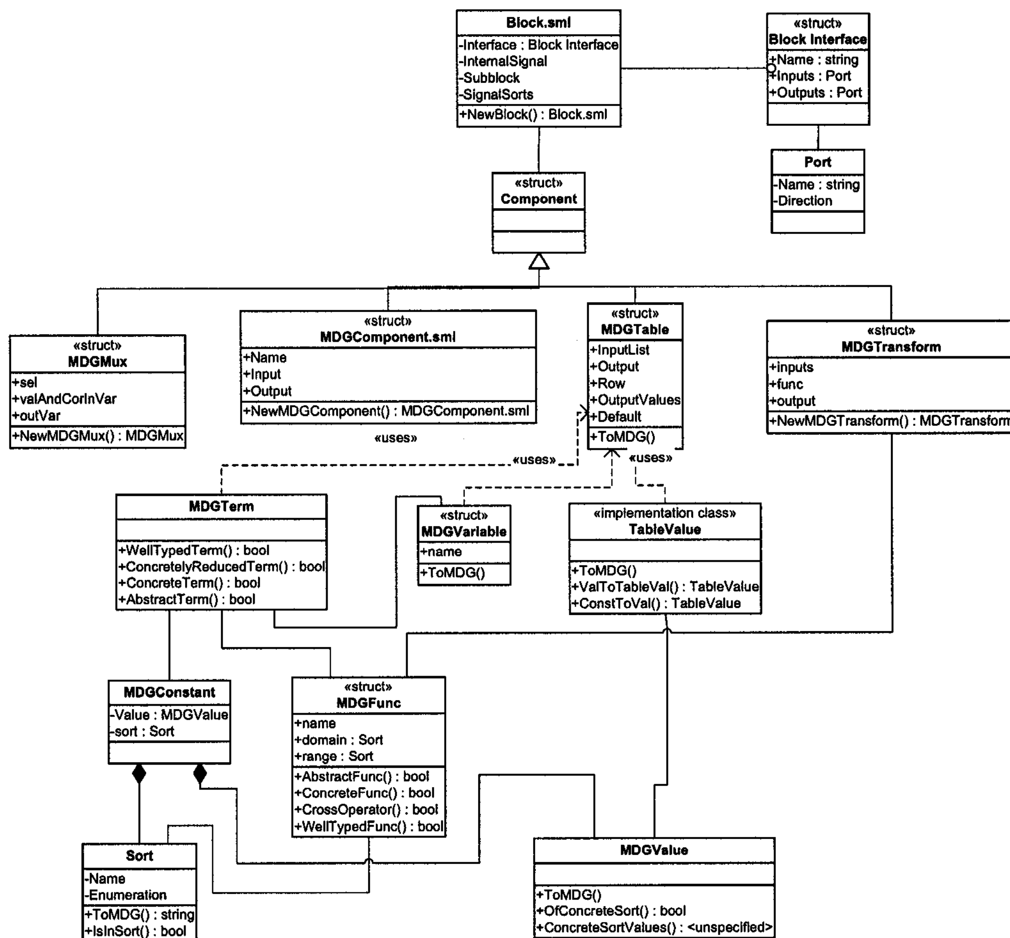


Fig 4.2: Class Diagram for the Data Structure

4.3 Issues Faced

While defining MDG variables, we first followed the embedding and described it as a combination of a name, a Boolean value and a sort. In the subsequent modules, it appeared to be cumbersome and unnecessary too, to define an MDG variable this way, in all the instances. This was mainly due to the common presence of the variables in the interface of all

the tables, multiplexer and transformation functions. So we had to modify the structure of variable so it constitutes out of its name only. In doing so, another issue surfaced- how to store the sort information of the variables when the particular component is a sub-block inside a block. To this end, we appended one more argument in the list of block's elements that would conserve the signals along with their respective sorts. This way, the chance of losing sort information was eliminated.

Another related issue was how to transport the sort of the signals from components to blocks when the component forms a sub block, owing mainly to the reason that, interface of a component does not contain MDG variables. As a solution to the problem, the tool would be empowered to determine the sorts of the input and output arguments depending on which component they belong to. For example, sorts of signals of a component that returns the OR-ed output of its two word inputs will be determined to be a word too.

4.4 Summary

In this chapter, we have discussed the design and implementation of the data structure with specific attention to the classes and modules of the design. Also, different issues that came along have also been highlighted. In order to apply and validate the proposed data structure, we introduce the design, specification and implementation of a single bus simple RISC processor. Further, the consequence of the application of the proposed structure on the processor design is reviewed and analyzed.

Chapter 5

Case Study

The previous chapter was all about the design and implementation of the data structure. We also elaborated different classes and modules of the structure. Now, we will focus on the description of Simple RISC processor followed by the formalization and implementation of the design.

5.1 Description of Simple RISC Processor

We selected a simple Reduced Instruction Set Computer (RISC) processor as our case study for the proposed hybrid tool due mainly to its generality and rich instruction set architecture. The subset of 12 instructions considered here are:

a. Load and store instructions:
   i. Direct and indexed addressing load: *ld*
   ii. Displacement addressing load: *la*
   iii. Register relative addressing load: *ldr*
   iv. Direct and indexed addressing store: *st*
b. Register relative addressing store: str Arithmetic instructions:
   i. 2's complement addition – *add*
   ii. Immediate 2's complement addition – *addi*
   iii. Negation – *neg*
c. Logical operations:
   i. Logical NOT – *not*
   ii. Logical OR – *or*
   iii. Immediate logical OR – *ori*
d. Branch operations: br

While modeling the processor, we followed the procedural steps that would take us from behavioral aspects of the controller to the point where structural details of the data path implementation were taken care of. The modeling starts in the form of specification of the behavior that the control system demonstrates. When the behavior of the controller and other sub modules of the control system are known, we can go on to define the interface of the control system. Descriptions of each input and each output form the interface. Finally the interface of data path, counter and memory system is determined.

5.1.1 Control System

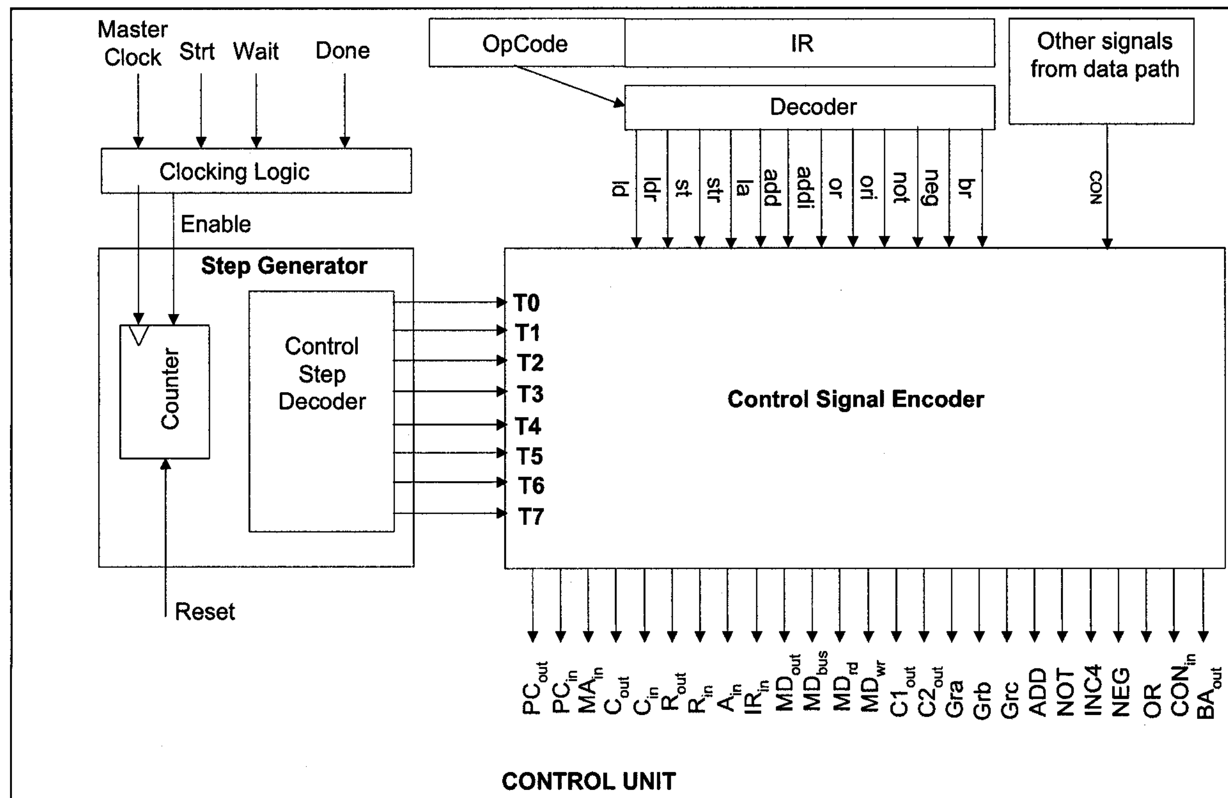


Figure 5.1 Control Unit

The control system contains a clocking logic, a control step generator and a signal encoder. The step generator is made up of counter and a control step decoder.

5.1.1.1 Control Signal Encoder

Control signal encoder is responsible for generation of a set of control signals that are used to drive data path modules. As input, the encoder receives control step signals, decoded OpCode and other signals from data path. Then the combinational circuit generates the corresponding control signal that will be the input of data path or memory.

5.1.1.2 Clocking Logic

Clocking logic receives 'Start' signal from external source, 'Run' signal from within the circuit, and 'Read', 'Write', and 'Stop' signal generated within the control unit.

The outputs are:

a. Enable: Signal that enables the counter and control step generator.
b. Rd/ Wr: In response of Read/Write signal from the control signal encoder, these signals are generated towards memory unit. The Rd/Wr signals remain latched until the corresponding memory operation is completed.

5.1.1.3 Control Step Generator

The control step generator is a synchronous parallel-load up counter in the design. It accepts an increment signal, the Master Clock in this case, an *enable* signal and a *Reset* signal that causes the counter to restart its step output at T0 at the next clocking event. The 3-bit counter counts from 0 to 7 when it is reset. For example, the *Reset* signal is generated by the *End* signal that terminates a control sequence. This unit generates control steps such as T0, T1....T7 corresponding to the sequential counter numbers 000, 001...111 etc. These control steps are input to the control signal encoder, which utilizes these steps along with other inputs from opcode decoder to generate appropriate control signals. These control signals are input to data path and memory units.

5.1.1.4 OpCode Decoder

The function of this unit is to read the opcode sitting in the first 5 bits of the instruction register (IR) and assert appropriate control signal (ld, ldr, st, str, la, add, addi, or, ori, not, neg, br) associated with the opcode towards control signal encoder. In essence, opcode decoder is a 5 to 1 decoder that depicts the instruction to be executed.

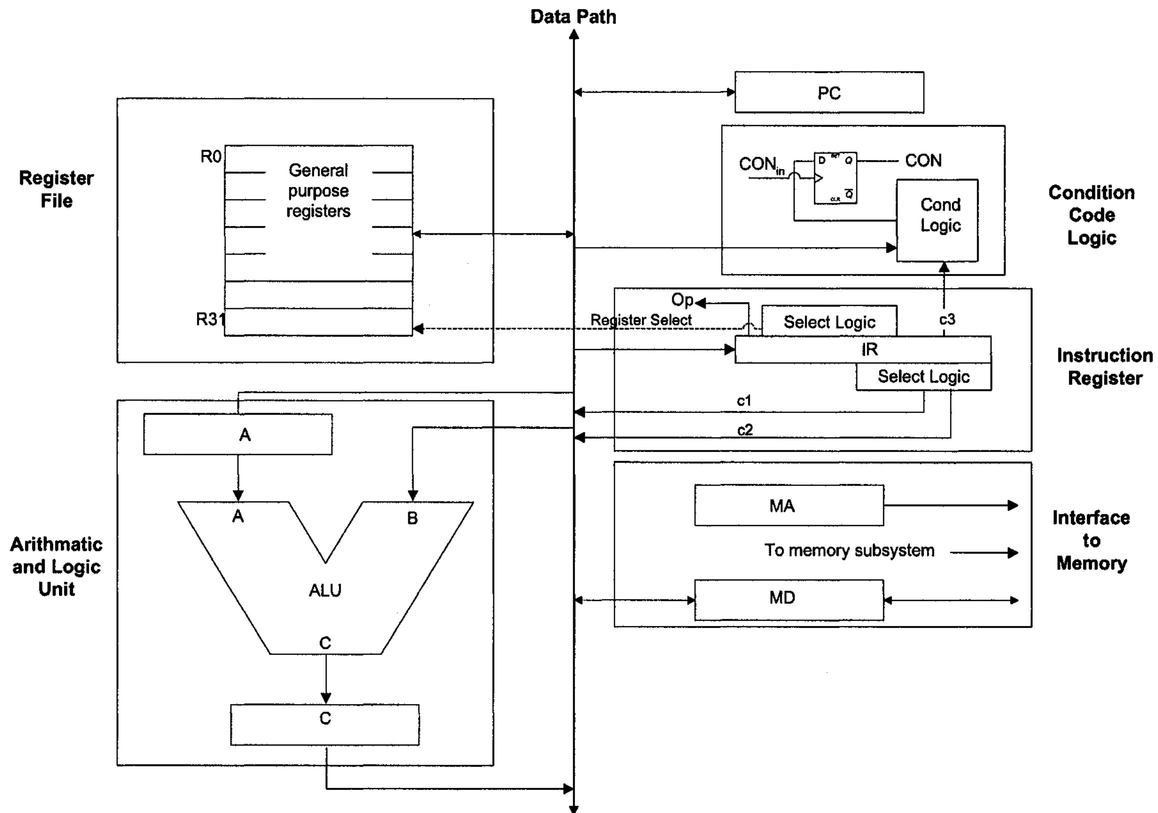


Figure 5.2 Data Path

5.1.2 Data Path

The design of the data path involves decisions at several levels of abstraction. The highest level is that of the micro architecture, where the basic registers and interconnections of an implementation are laid out. There are also design decisions to be made about the type of flip-flop used to implement registers and the type of interconnecting bus to use. In general, the high-level decisions affect the register transfers that can be performed and how many of

them can be performed simultaneously. The low-level decisions affect the speed at which each register transfer can be done. There is a limited influence of the flip-flop type on the kind of register transfer that can be done.

Data path is made up of Register file, Arithmetic Logic Unit (ALU), Condition Code Logic unit, Instruction Register (IR), and Interface to memory. Program Counter (PC) points to the next instruction to be executed and the IR holds the current instruction. Three control signals- PC_{out}, PC_{in} , and IR_{in} are associated to PC and IR. 5-bit register select and three other fields- *c1*, *c2* and *c3* – are extracted from IR. Registers MA (memory address) and MD (memory data), along with their associated signals, provide the interface to the memory system. MA receives the memory address from the bus through the unidirectional channel, whereas, MD is bidirectional in the sense that it receives data from memory and transfer it to bus during a 'read' operation and vice versa during a 'write'.

5.2 Formalization

As we completed defining the interface, the formalization of the specification can proceed.

5.2.1 Main Modules and Their Design

As the inputs and outputs of the control unit are determined and defined, derivation of Boolean expressions is the next step. In order to be able to analyze the control signals and derive respective Boolean expressions, the instruction formats deserves an examination.

5.2.1.1 Instruction Set Architecture:

As shown in figure 5.3, all instructions are 32 bits long. The memory operands can be accessed through load and store instructions only in SRC, as they are load-store class of

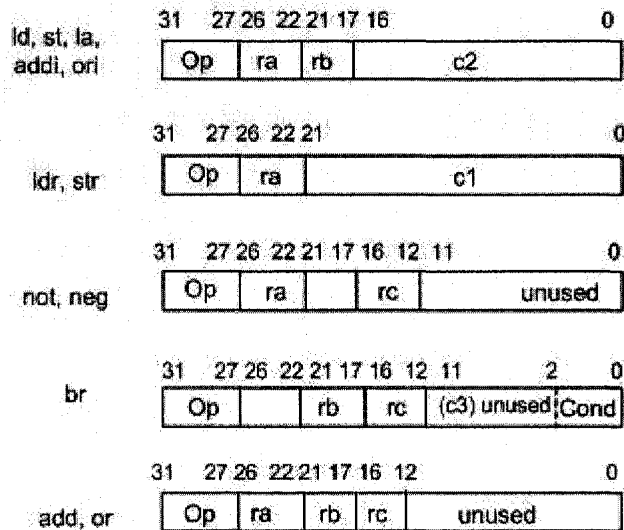


Fig 5.3 : Instruction Formats

machine. Opcode is 5 bits long. So, a total of $2^5 = 32$ different instructions are possible. Here we consider 12 of these instructions. The ra, rb and rc are 5 bit fields which point to one of 32 general purpose registers (GPR).

c1 is a 22 bit field that contains one of the following:

1. Constant to be added to Program Counter (PC) in order to calculate the address of the memory data to be loaded to a GPR

Example: *ldr ra, c1* : Load Register relative : R[ra] = M[PC + c1]

2. Constant to be added to Program Counter (PC) in order to calculate the address of the memory location where content of a GPR will be stored

   Example: *str ra, c1* : Store Register relative : M[PC + c1] = R[ra]

c2 holds a 17 bit value. In case when the value contained in rb is 0, which points to r0, the control unit recognizes that c2 holds the sign extended 2's compliment number that specifies the address itself. c2 contains one of the following:

1. Memory address of the memory data that will be loaded to a general purpose register – Example: ld ra, c2 : Direct addressing: R[ra] = M [c2]
2. Constant to be added to the content of a GPR to calculate the address of the memory data that will be loaded to another GPR.

   Example: ld ra, c2 (rb) : Indexed addressing R[ra]= M[c2 + R[rb]]
3. Memory address where the content of a GPR will be stored

   Example: st ra, c2 : Direct addressing: M [c2] = R[ra]
4. Constant to be added to the content of a GPR to calculate the memory address where the content of another GPR will be stored.

   Example: st ra, c2 (rb) : Indexed addressing M[c2 + R[rb]] = R[ra]
5. Constant, acting as the displacement address argument, to be loaded to a GPR

   Example: la ra, c2: Load displacement address: R[ra] = c2
6. Constant, to be added to be added to the content of a GPR in order to calculate the displacement address, to be loaded to another GPR

   Example: la ra, c2 (rb): Load displacement address: R[ra] = c2 + R[rb]

c3 ranges over the 3 least significant bits of the 32 bit instruction, allowing to specify one of 0,1,2,3,4 or 5. It contains the condition that is tested against the contents of a GPR to determine branch operations.

Example: br rb, rc, c3: Branch to R[rb] if R[rc] fulfills condition in c3

op<4..0> := IR <31..27>: Operation code field

ra<4..0> := IR <26..22>: Target register field

rb<4..0> := IR <21..17>: Operand, address index or branch target register

rc<4..0> := IR <16..12>: Second operand, conditional test

c1<21..0> := IR <21..0>: Long displacement field

c2<16..0> := IR <16..0>: Short displacement or immediate field

c3<11..0> := IR <11..0>: Modifier field

Branch conditions are dependent on the 'cond' field, c3<2..0>, and conditional branches are dependent on the value of a general register rather than the value of a flag or flags register:

cond :=c3 <2..0> = 0 → 0:	Never
c3 <2..0> = 1 → 1:	Always
c3 <2..0> = 2 → R[rc] = 0:	If register is zero
c3 <2..0> = 3 → R[rc] ≠ 0:	If register is nonzero
c3 <2..0> = 4 → R[rc]<31> = 0:	If positive or zero
c3 <2..0> = 5 → R[rc] <31> =1:	If negative

5.2.2 Register Transfer Notation (RTN)

Although plain English is useful for describing common features of a machine design and its general capabilities, industrial or research based design of a computer can not rely on that view. In pursuit of such designs, precise specifications of functions are indispensable. Register transfer language facilitates the precise specification of the transfer of data among registers and memory cells. Register transfer notation, thus, is used to describe the components and operation of SRC.

An example of RTN for the instruction ldr ra, c1 is given in table 5.1.

Opcode 2   ldr ra, c1

Control step	RTN	Control sequence
T0	MA ← PC: C ← PC+4;	PC_{out}, MA_{in}, INC4, C_{in}
T1	MD ← M[MA]: PC ← C;	Read, C_{out}, PC_{in}, Wait
T2	IR ← MD;	MD_{out}, IR_{in}
T3	A ← PC;	PC_{out}, A_{in}
T4	C ← A + c1;	$c1_{out}$, Add, C_{in}
T5	MA ← C;	C_{out}, MA_{in}
T6	MD ← M[MA]	Read, Wait
T7	R[ra] ← MD;	MD_{out}, Gra, R_{in}, End

Table 5.1 Register Transfer Notation

Formalization of the processor design goes through the following steps:

1. Translate Boolean expressions into MDG tables
2. Specification of the controller behavior in HOL
3. Specification of data path in HOL
4. Formulation of structural model of the controller and data path

Standard logic manipulations of the logic expressions provide us enough information to form tables for each control signal. However, in order to keep the table size small and for the sake of simplicity, we defined some concrete sorts to group signals together. In the process, we grouped all the memory accessing signals- ld, ldr, st, str, la- into the sort, "s_ldst". Similarly, Arithmetic operation signals- add, addi, or, ori, not, neg, inc4-, branch operation signal- br-, and condition operation signal –con- and control steps- T0, T1, T2, T3, T4, T5, T6 and T7- have been bundled in the sorts, "s_arithop", "s_brop", "s_cndlgc" and "s_ctlstp" respectively. One example of the resulting tables is as shown in table 5.2. From the table, we can conclude the following Boolean expression for PC_{out} :

$PC_{out} = T0 + T3.(ldr + str)$

All the tables required for the formalization process have been furnished in Appendix A.

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	PC_{out}
T0	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	Ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	Str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

Table 5.2 Truth table for control signal, PC_{out}

The MDG embedding in HOL contains a 'TABLE' construct that allows us to define the encoded controller signals as outputs of those tables that take the individual signals, that come as input to the encoder, as its inputs. An example of such a table construct, written for 'PC_{out}', in HOL is given below:

```
⊢ PC_out_tab (ctlstp,ldst) PC_out
= ~(stateVariable PC_out) /\
  TABLE [vtt ctlstp;vtt ldst] PC_out
   [[itv 0;DONT_CARE]; [itv 3;itv 2]; [itv 3;itv 3]]
   [btc T;btc T;btc T] (btc F)
```

As described earlier, Table has a list of inputs as MDG Terms, an output as MDG Variable, a list of input value lists as Table values, a list of corresponding output values as MDG Terms and a default value again as MDG Term. A predicate, stating that the output is not a state variable, is put in conjunction with the table itself. 'vtt', 'itv', 'btc' etc. are casting functions required for transforming the values into warranted forms. 'DON'T_CARE' is a value assigned to an input variable when it does not affect the value of output variable.

All the output signals of control step encoder are defined in this way. Then, the top-level encoder is obtained as a conjunction of the interfaces of all the signal definitions. The interface of the encoder contains all the inputs and outputs as shown in figure 5.1.

Designing counter includes conjunction of predicates that declare the outputs of the counter as state variables. The counter input is an asynchronous reset. In each clock cycle, the counter increments its output by 1 until the output equals 7 after which it restarts counting from 0 again. At any point, the asynchronous reset can initialize the counter.

The control step generator is largely a 3 to 8 decoder. Its inputs are the outputs of the counter and it generates the control steps T0, T1...etc. provided the 'enable' signal originated from the clocking logic is set.

The clocking logic inputs include external inputs such as 'start', internally generated signals such as 'wait' and 'done', 'Read' and 'Write' signals from control signal encoder. It sends 'enable' to the control step generator, and 'read' and 'write' latching signals to memory system.

Data path modules- ALU, Register file, Instruction register, Memory interface, and condition code logic- are defined separately in HOL. While defining ALU, inputs and output are considered as words. The output values of the table are un-interpreted function symbols according to the requirement of MDG. To this end, ALU table dictates which function symbol, from among 'add', 'or', 'not', 'neg' and 'inc4', should be the value of the output depending on the value of the control signals- ADD, OR, NOT, NEG and INC4. As these are words, the bus width of the inputs and the output are also taken into account. The definition also includes a conjunction of a predicate that warrants that the bus widths of all the inputs are equal.

The top-level of data path design is again a conjunction of the instances of individual modules along with predicates and declaration of internal signals etc.

5.2.3 Implementation

Implementation of both the controller and the data path are structural models. They represent a net list of components. The set of components is made up of subsets of components defined in the MDG embedding in HOL, and subsets of user defined components.

5.2.3.1 Controller

Structural definition of the controller involves simply a realization of the logic equations derived earlier. Using logic gates defined in the embedding, we completed the net list form of the equations. In order to keep the definitions of individual control signals small, we grouped the inputs of the control signal encoder according to their functionalities. For example, we defined 'ld_ldr_gr', which is an instantiation of a 2-input OR gate with 'ld' and 'ldr' as its inputs and 'ld_ldr' as its output. Thus, whenever an OR-ed form of 'ld' and 'ldr' would be required in any of the succeeding definitions, we would be able to use 'ld_ldr' rather than defining the sub module repeatedly.

As described before, control signal encoder is a combinational circuit that is hard-wired to provide the desired control signal outputs according to the logic equations. To this end, the structural model is devoid of any sequential element such as registers. However, the counter is implemented with registers besides other components.

Same as the specification, implementation of the top-level controller is the conjunction of instances of control signal encoder, clocking logic and control step generator.

5.2.3.2 Data Path

A register with control is defined in the components. Such a component is instantiated in the implementation of sequential sub modules in data path. An example of such a sub module is the register 'A' that holds the operand for ALU. Similar to the controller, data path top-level model is the conjunction of the instances of sub-modules that constitutes its structure. The ALU operands are implemented as words. However, the embedding allows us to implement them as integers too.

5.3 Application Results and Analysis

As a verification tool, MDG accepts MDG-HDL which allows the use of abstract variables for representing data signals. The MDG-HDL is then compiled into the ASM model in internal MDG data structures. The simple RISC processor has been specified and implemented using HOL. In this case the MDG embedding in HOL served the purpose of MDG-HDL. Besides specification and implementation a parser has also been designed and implemented. The main function of the parser is to parse the circuit definitions in structural and behavioral terms.

We used the parser for parsing the description of specification and implementation of the SRC processor. The specification is generally a behavioral description of the circuit. It uses tabular representation of input and output relations. The proposed data structure supports the use of abstract function symbols as output variables in the table. As stated earlier, the embedding allows us to describe the circuit as a list of theorems in HOL. The parser parses these theorems as statements. A function, 'StatToTuple', defined in the ParserSupport module, disintegrates this list of statement into a tuple of signals, their sort assignments,

components, outputs and the mapping between state variables and next state variables which are the required inputs of MDG tool.

Signals are parsed into a tuple containing the signal identifier and its associated sort assignment. For example, *signal (CON, bool)* represents a signal 'CON' which is of type Boolean.

Components are parsed following the MDG tool syntax as *Component (CompName, CompDef)*, where CompName is the component identifier and CompDef represents its definition. A component is generally defined by instantiating the input/output ports of a predefined component module in MDG. In addition to this, our data structure allows a component to be a table, or a user-defined component. We also incorporated an additional field in the interface of a block. When a user defines a list of components in conjunction among themselves, we term that as a grammar block. Both a block and a grammar block contain a field in their interfaces that lists the signals and their corresponding sort information. Consequently when a block is extracted from a parsed module, the signals and their sorts are readily available for extraction by the tool. As we see, when the design specification and implementation is parsed, the compiled output of the parser is in the form as warranted by MDG. So, when the proposed tool calls MDG for verification of a block that is sufficiently tractable, it finds the required circuit description file. As rests of the required files are generated by the proposed tool, the MDG tool can go ahead with different stages of verification.

5.3.1 Verification of an ALU:

In place of the MDG-HOL hybrid tool to verify the proposed data abstraction structure, we validated an ALU in HOL. In this case the ALU was formed to comply with the proposed data structure in terms of abstraction and refinement.

The ALU comprised two functions- addition and multiplication. The behavioral model contains a table that directs to the desired output function according to the inputs. The structural implementation uses components for transformation and multiplexing the inputs towards one expected output.

The verification process advances in stages. Firstly, an overall proof goal was set to prove that the structural ALU model implies the behavioral specification. Secondly, because the goal was too deep to prove, it was disintegrated into two sub goals to prove that the components used in the implementation are the implication of the table that depicts the behavior of the ALU and the predicate relating to the state variable. Proof of each sub goal involves proving the member theorems top down recursively. Thus, when the two sub goals are proved, the over all goal is verified. The verbose messages of the proof procedure are furnished in Appendix B.

The main difference between this verification and a similar procedure with the existing hybrid tool would be in the data size. The existing MDG-HOL hybrid tool verifies the model assuming that the operands are one bit long. The proposed data structure facilitates the verification of the ALU with the operands defined as words. Thus it improves upon the data handling capacity and approaches towards practical applications.

5.4 Summary

The objective in this chapter was to utilize the processor model so the proposed and implemented data structure could be applied on it and the consequent result would be analyzed. Finally, by using HOL, we verified an ALU that uses the proposed data structure. Next chapter sums up the findings so far, points to the probable future research direction and wraps up the thesis.

Chapter 6

Conclusion and Future Work

In the last chapter, we shed focus on the case study relating to the Simple one-bus RISC Processor. Also, we furnished the result and analysis of the application of the proposed data abstraction structure on the processor specification and implementation. In this chapter we will conclude our findings and also indicate the future research prospects.

Automatic tools are convenient to use while they lack in ability to handle practical circuits. The main reason behind this is the state space explosion problem associated to these state space exploration based techniques. Interactive techniques are capable of handling large, industrial designs. However, their interactive feature warrants for user involvement and expertise, which is again unsuitable for industrial purposes. Concocted solutions of these two approaches are under researchers' scrutiny for quite a while.

One major aspect that invites swelling of design size is the data structure involved in the data path operations. Refined data structures at bit level increases the data width requirements exponentially. With the same data represented in a more abstract level, subsequent upsurge of resultant data size can be reduced considerably and thus the capacity of the verification tool is enhanced largely. Our focus has been on this approach of optimization.

6.1 Major Contributions

An MDG embedding in HOL is the embedding of MDGs as built-in data types. We extended the existing embedding in order to incorporate abstract sorts and related functions. MDG operations are interfaced to HOL functions. This allows the tool to maneuver graphs rather

than corresponding HOL terms. For supporting the abstract data sort structure, new data types and functions have been included in the embedding. Then we designed and implemented a simple RISC processor incorporating abstract data sorts, defined in the embedding, in the controller and data path circuits. The main intention behind this was to apply the data abstraction structure to the design of a circuit and assimilate the forthcoming design of the tool that integrates the proposed data abstraction structure. With this view in mind, we also designed and implemented a parser that successfully parses both the specification and implementation of the processor. This would work as the parser for the main module of the proposed verification tool that extracts blocks, sub blocks and grammar blocks from the parsed modules.

The next step involved the design of the modules, acting as the building blocks of the proposed data structure, that encompass the sorts, values, variables, terms, constants, functions, components and tables. Also included in these modules are methods used to transform signals, terms, variables, functions etc. into MDG-formats, to check the well-typed ness, to check other predicates etc.

6.2 Future Research Directions

With the existing data structure in place, the MDG-HOL hybrid tool would allow the specification and implementation to range over different data abstraction levels starting from Boolean, concrete towards integer and abstract sorts. In addition, Boolean words offer an intermediate abstraction level.

The existing hybrid tool uses three tactics- two for dealing with combinational and sequential circuits and the other for managing hierarchical verification. The proposed tool, which will

utilize the implemented data abstraction structure as its backbone, would exercise the full capacity when more tactics are designed and implemented. These tactics would allow the tool designer elaborate the specification and implementation of the design under test at different abstraction levels and deal with un-synthesizable data sorts.

Finally, if the future tool includes a quantitative direction to the user that enables to determine whether or not the design under verification needs abstraction in order to be verified by automated tool, it would make it more attractive to the industrial applications.

Bibliography

1. W. E. Dijkstra, *The Humble Programmer*, ACM Turing Award Lecture, 1972.
2. N. Shankar, *Combining Theorem Proving and Model Checking through symbolic Analysis,* CONCUR, 2000.
3. S. S.Kort, S.Tahar, P. Curzon, *Hierarchical Verification Using an MDG_HOL Hybrid Tool*, International Journal on Software Tools for Technology Transfer, Vol. 4, Springer Verlag, 2002, pp. 1-10.
4. G. S. Kumar, *Slides on theorem prover,* Center for Formal Design and Verification of Software, IIT, Bombay, India, 2002.
5. S. Berezin. *Model Checking and Theorem Proving: a Unified Framework.* Ph.D. Thesis, Carnegie Mellon University, 2002.
6. E. Clarke, A. Biere, R. Raimi, and Y. Zhu, *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design, Vol. 19, issue 1, July 2001.
7. E.M. Clarke, O. Grumberg, and D.E. Long. *Model checking and abstraction.* ACM Transactions on Programming Languages and Systems, Vol. 16(5), pp. 1512--1542, September 1994.
8. E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*, Cambridge: MIT Press, 1999.
9. M.H. Zobair and S. Tahar, *Formal Verification of a SONET Telecom System Block;* Proc. International Conference on Formal Engineering Methods (ICFEM'02), Shanghai, China, October 2002.

10. V.K. Pisini, S. Tahar, O. Aït-Mohamed, P. Curzon, and X. Song. *An Approach to Link HOL and MDG for Hardware Verification*. Proc. of the 1999 Micronet Workshop, Ottawa, Canada, April 1999, pp. 156-157.

11. S. Tahar, E. Cerny and X. Song, *Slides on Formal Verification of Systems,* Department of Electrical and Computer Engineering, Concordia University, Montréal, Canada, 1999.

12. Fabio Somenzi. "Binary Decision Diagrams". In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design,* NATO Science Series F: Computer and Systems Sciences, Vol. 173, pp. 303–366. IOS Press, 1999.

13. Y. Lakhnech, S. Bensalem, S. Berezin, S. Owre, *Incremental Verification by Abstraction,* Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001

14. John Harrison, Verifying Floating-Point Algorithms using Formalized Mathematics, Presentation, HVG Concordia, 2005

15. L. Paulson. *ML for the Working Programmer*, Cambridge University Press, 1996.

16. O. Parshin, *Specification and verification of the ARM6 microprocessor in HOL*, State of the Art of Formal Hardware Verification Seminar, January 2004.

17. P. Curzon, S. Tahar, O. Ait Mohammed, *Verification of the MDG Components Library in HOL.* In Jim Grundy and Malcolm Newey, editors, Theorem Proving in Higher Order Logics: Emerging Trends, Department of Computer Science, The Australian National University, pp. 31-46, 1998.

18. R.Bryant, *Graph-based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, Vol. C-35(8) pp. 677-691, August 1986.

19. V.K.Pisini, S.Tahar, P.Curzon, O. Ait-Mohamed and X. Song. *A hybrid approach to formal verification using HOL and MDG, 1999.*

20. Z. Zhou, N. Boulerice, *MDG Tools User's Manual*, 1996

21. T. Mhamdi and S. Tahar: *Embedding Multiway Decision Graphs in HOL*; B-Track Proc. International Conference on Theorem Proving in Higher-Order Logics (TPHOLs'04), Park City, Utah, USA, pp. 121-136, September 2004.

22. Y.Xu, X. Song, E. Cerny, O. Ait Mohamed, *Model Checking for a First-Order Temporal Logic Using MDGs*, The Computer Journal, The British Computer Society, 2004.

23. M. Newey, *Mechanical Verification,* Lectures in The Department of Computer Engineering, Australian National University, 2000

24. S. Rajan, N. Shankar, M.K. Srivas. *An integration of model checking with automated proof checking.* In Pierre Wolper, editor, *Computer-aided Verification*, CAV '95, vol. 939 of *Lecture Notes in Coputer Science*, pp 412-416, Passau, Germany, March 1996, Springer-Verlag.

25. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts,1993.

26. G. Holzmann. *The model checker SPIN.* IEEE Transactions on Software Engineering, vol. 23(5), pp. 279–295, May 1997.

27. R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Rajeev Alur and Thomas A. Henzinger,editors, Conference on Computer Aided Verification (CAV), vol. 1102 of *LectureNotes in Computer Science*, pp. 423–427, New Brunswick, NJ, USA, July/August 1996.Springer Verlag.

28. A. Aziz, F. Balarin, S.-T. Cheng, R. Hojati, T. Kam, S.C. Krishnan, R.K. Ranjan, T.R. Shiple,V. Singhal, S. Tasiran, H.-Y. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. HSIS:*A BDD-Based Environment for Formal Verification.* In ACM/IEEE Design Automation Conference(DAC), San Diego, CA, June 1994. San Diego Convention Center.

29. R. K. Brayton, A. L. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri,Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel,F. Somenzi, A. Pardo, and S. Sarwary. VIS: *A system for verification synthesis*. In Computer-Aided Verification, New Brunswick, NJ, July-August 1996.

30. M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. *Automatic Verification of Sequential Circuits Using Temporal Logic*. IEEE Transactions on Computers, vol. C-35(12), pp. 1034–1044,December 1986.

31. D.L. Dill and E.M. Clarke. *Automatic verification of asynchronous circuits using temporal logic*. IEE Proceedings, vol. 133 Part E(5), pp. 276–282, September 1986.

32. E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. *Verification of the Futurebus+ Cache Coherence Protocol.* In D. Agnew, L. Claesen, and R. Camposano, editors, IFIP Conference on Computer Hardware Description Languages.

33. F. Corella, Z. Shou, X. Song, M. Langevin and E. Cerny. *Multiway Decision Graphs for Automated Hardware Verification.* Formal Methods in System Design, Vol 10 (1), pp 7-46, 1997.

Appendix A

LOGIC EXPRESSIONS FOR CONTROL SIGNLAS

PC_{out} = T0 + T3.(ldr + str)

PC_{in} = T1 + T4.br.CON

MA_{in} = T0 + T5.(ld + st + ldr + str)

C_{out} = T1 + T4.(not + neg) + T5.(ld + ldr + st + str + la + add + addi + or + ori)

C_{in} = T0 + T3.(not + neg) + T4.(ld + ldr + st + str + la + add + addi + or + ori)

R_{out} = T3.(not + neg + add + addi + or + ori + br) + T4.(add + or + br) + T6.(st + str)

R_{in} = T4.(neg + not) + T5.(add + addi + or + ori + la) + T7.(ld+ldr)

A_{in} = T3.(ld +ldr + st + str + add + addi + or + ori + la)

IR_{in} = T2 + T6.(ld + ldr)

MD_{out} = T2 + T7.(ld + ldr)

MD_{bus} = T6.(st + str)

MD_{rd} = T1 + T6. (ld + ldr)

MD_{wr} = T7.(st + str)

$C1_{out}$ = T4.(ldr + str)

$C2_{out}$ = T4.(ld +st + la + addi + ori)

Gra = T4.(not + neg) + T5.(la + add + addi + or + ori) + T6.(st + str) + T7.(ld + ldr)

Grb = T3.(ld + la + st + add + addi + or + ori) + T4.br

Grc = T3.(neg + not + br) + T4.(add +or)

ADD = T4.(ld + ldr + la + st + str + add + addi)

NOT = T3.not

INC4 = T0

NEG = T3.neg

OR = T4.(or + ori)

CON_{in} = T3.br

READ = T1 + T6.(ld + ldr)

WRITE = T7.(st + str)

WAIT = T1 + T6 (ld + ldr)

BA_{out} = T3.(ld + st + la)

END = T4.(neg + not + br) + T5.(la + add + addi + or + ori) + T7.(ld + ldr + st + str)

<u>CONCRETE SORTS</u>

Ctrl_Stp	:	T0, T1, T2, T3, T4, T5, T6, T7
Ld_St	:	ld, ldr, la, st, str
Arith_Op	:	add, addi, or, ori, not, neg
Br_Op	:	br
Cond_Log	:	CON

TRUTH TABLES FOR CONTROL SIGNALS

PC_{out}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	PC_{out}
T0	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	Ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	Str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

PC_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	PC_{in}
T1	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	DON'T_CARE	br	CON	1
Default					0

MA_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	MA_{in}
T0	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

C_{out}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	C_{out}
T1	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T5	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	add	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	addi	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	or	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T5	ori	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

C_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	C_{in}
T0	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T4		add	DON'T_CARE	DON'T_CARE	1
T4		addi	DON'T_CARE	DON'T_CARE	1
T4		or	DON'T_CARE	DON'T_CARE	1
T4		ori	DON'T_CARE	DON'T_CARE	1
T4	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

R_{out}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	R_{out}
T3	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
T4	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
T6	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

R_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	R_{in}
T4	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
T5	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

A_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	A_{in}
T3	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
T3	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

IR_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	IR_{in}
T2	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

MD_{out}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	MD_{out}
T2	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

MD_{bus}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	MD_{bus}
T6	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

MDrd

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	MD_{rd}
T1	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

MD_{wr}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	MD_{wr}
T7	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

$C1_{out}$

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	$C1_{out}$
T4	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

$C2_{out}$

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	$C2_{out}$
T4	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
Default					0

Gra

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	Gra
T4	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
T5	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

Grb

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	Grb
T3	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
Default					0

INC4

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	INC4
T0	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

Grc

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	Grc
T3	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
T3	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
T4	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
Default					0

ADD

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	ADD
T4	DON'T_CARE	add	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	addi	DON'T_CARE	DON'T_CARE	1
T4	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T4	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

NOT

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	NOT
T3	DON'T_CARE	not	DON'T_CARE	DON'T_CARE	1
Default					0

NEG

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	NEG
T3	DON'T_CARE	neg	DON'T_CARE	DON'T_CARE	1
Default					0

OR

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	OR
T4	DON'T_CARE	or	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	ori	DON'T_CARE	DON'T_CARE	1
Default					0

CON_{in}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	CON_{in}
T3	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
Default					0

READ

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	READ
T1	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

WRITE

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	WRITE
T7	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

WAIT

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	WAIT
T1	DON'T_CARE	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T6	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

BA_{out}

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	$C2_{out}$
T3	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T3	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

END

Ctrl_Stp	Ld_St	Arith_Op	Br_Op	Cond_Log	R_{in}
T4	DON'T_CARE	Not	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	Neg	DON'T_CARE	DON'T_CARE	1
T4	DON'T_CARE	DON'T_CARE	br	DON'T_CARE	1
T5	DON'T_CARE	Add	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	Addi	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	Or	DON'T_CARE	DON'T_CARE	1
T5	DON'T_CARE	Ori	DON'T_CARE	DON'T_CARE	1
T5	la	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ld	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	ldr	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	st	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
T7	str	DON'T_CARE	DON'T_CARE	DON'T_CARE	1
Default					0

Appendix B

VERBOSE MESSAGES OF PROOF PROCEDURE

```
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "opcode_def".
> val opcode = |- opcode = CONCRETE "opcode" ["add"; "mul"] : thm
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "ALU_beh_def".
> val ALU_beh =
   |- !intrp assgn code x y z.
       ALU_beh intrp assgn (code,x,y) z =
       ~stateVariable z /\
       TABLE intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
         [add_t x y; mul_t x y] (add_t x y) : thm
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "ALU_str_def".
> val ALU_str =
   |- !intrp assgn code x y z.
       ALU_str intrp assgn (code,x,y) z =
       ?sig_add sig_mul.
         mdg_transform intrp assgn ([x; y],mdg_add) sig_add /\
         mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul /\
         mdg_mux assgn
           (code,
            [(string_to_val "add",sig_add);
             (string_to_val "mul",sig_mul)]) z : thm
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "FA_str_def".
> val FA_str =
   |- !assgn a b cin s cout.
       FA_str assgn (a,b,cin) (s,cout) =
       ?p g sig.
         mdg_xor assgn (a,b) p /\ mdg_and assgn (a,b) g /\
         mdg_and assgn (p,cin) sig /\ mdg_xor assgn (p,cin) s /\
         mdg_or assgn (g,sig) cout : thm
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "wires_4_def".
> val wires_4 =
   |- !assgn x x0 x1 x2 x3.
       wires_4 assgn x x0 x1 x2 x3 =
       wire assgn x 0 x0 /\ wire assgn x 1 x1 /\ wire assgn x 2 x2 /\
       wire assgn x 3 x3 : thm
<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "RCA4_str_def".
```

```
> val RCA4_str =
   |- !assgn a b cin s cout.
       RCA4_str assgn (a,b,cin) (s,cout) =
       ?a0 a1 a2 a3 b0 b1 b2 b3 c0 c1 c2 s0 s1 s2 s3.
         wires assgn a [a0; a1; a2; a3] /\
         wires assgn b [b0; b1; b2; b3] /\
         wires assgn s [s0; s1; s2; s3] /\
         FA_str assgn (a0,b0,cin) (s0,c0) /\
         FA_str assgn (a1,b1,c0) (s1,c1) /\
         FA_str assgn (a2,b2,c1) (s2,c2) /\
         FA_str assgn (a3,b3,c2) (s3,cout) : thm
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd>>
Definition has been stored under "RCA4_beh_def".
> val RCA4_beh =
   |- !assgn a b cin s cout.
       RCA4_beh assgn (a,b,cin) (s,cout) =
       !t.
         ?sw.
           (sw =
            NBWORD 5
              (BNVAL (val_to_word (assgn a t)) +
               BNVAL (val_to_word (assgn b t)))) /\
           (MDG_value_to_bool (assgn cout t) = MSB sw) /\
           (val_to_word (assgn s t) = WSEG 4 1 sw) : thm
> val assum = fn : int -> term
> val UNDISCH_N_TAC = fn :
  int -> term list * term -> (term list * term) list * (thm list -> thm)
> val RW_HYP_TAC = fn :
  thm list -> term list * term -> (term list * term) list * (thm list -> thm)
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
   Proof manager status: 1 proof.
   1. Incomplete:
       Initial goal:
       !intrp assgn code x y z q.
         ALU_str intrp assgn (code,x,y) z ==>
         ALU_beh intrp assgn (code,x,y) z


    : proofs
OK..
2 subgoals:
> val it =
   TABLE intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y)
   ------------------------------------
```

```
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z

   ~stateVariable z
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z
    : goalstack
OK..
Meson search level: ..

Goal proved.
 [...] |- ~stateVariable z

Remaining subgoals:
> val it =
   TABLE intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y)
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z
    : goalstack
OK..
2 subgoals:
> val it =
   table intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z
```

```
   compatible_assgn assgn
   ------------------------------------
     0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
     1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
     2. mdg_mux assgn
          (code,
           [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
          z
    : goalstack
OK..
Meson search level: ..

Goal proved.
 [...] |- compatible_assgn assgn

Remaining subgoals:
> val it =
   table intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
     0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
     1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
     2. mdg_mux assgn
          (code,
           [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
          z
    : goalstack
OK..
2 subgoals:
> val it =
   outTable intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
     0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
     1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
     2. mdg_mux assgn
          (code,
           [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
          z
     3. ~stateVariable z

   transitionTable intrp assgn [vtt code] z
     [[conctv "add"]; [conctv "mul"]] [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
     0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
```

```
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z
    3. stateVariable z
   : goalstack
OK..
Meson search level: ..

Goal proved.
 [....]
|- transitionTable intrp assgn [vtt code] z
    [[conctv "add"]; [conctv "mul"]] [add_t x y; mul_t x y] (add_t x y)
    t

Remaining subgoals:
> val it =
   outTable intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. mdg_mux assgn
         (code,
          [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
         z
    3. ~stateVariable z
   : goalstack
OK..
1 subgoal:
> val it =
   mdg_mux assgn
     (code,[(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
     z ==>
   outTable intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. ~stateVariable z
   : goalstack
OK..
3 subgoals:
> val it =
   assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))
```

```
------------------------------------
 0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
 1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
 2. ~stateVariable z
 3. ~(assgn code t = CONC_VAL "add")
 4. ~(assgn code t = CONC_VAL "mul")
 5. compatible_assgn assgn
 6. concreteVariable code
 7. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
 8. list_holds_all (ofConcreteSortSym (sortOfVariable code))
     [CONC_VAL "add"; CONC_VAL "mul"]
 9. list_holds_all (var_has_sort (sortOfVariable z))
     [sig_add; sig_mul]
 10. !t.
      (if assgn code t = CONC_VAL "add" then
        assgn z t = assgn sig_add t
       else
        (assgn code t = CONC_VAL "mul") /\
        (assgn z t = assgn sig_mul t))

assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t))
------------------------------------
 0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
 1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
 2. ~stateVariable z
 3. ~(assgn code t = CONC_VAL "add")
 4. assgn code t = CONC_VAL "mul"
 5. compatible_assgn assgn
 6. concreteVariable code
 7. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
 8. list_holds_all (ofConcreteSortSym (sortOfVariable code))
     [CONC_VAL "add"; CONC_VAL "mul"]
 9. list_holds_all (var_has_sort (sortOfVariable z))
     [sig_add; sig_mul]
 10. !t.
      (if assgn code t = CONC_VAL "add" then
        assgn z t = assgn sig_add t
       else
        (assgn code t = CONC_VAL "mul") /\
        (assgn z t = assgn sig_mul t))

assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))
------------------------------------
 0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
 1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
 2. ~stateVariable z
```

```
    3. assgn code t = CONC_VAL "add"
    4. compatible_assgn assgn
    5. concreteVariable code
    6. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    7. list_holds_all (ofConcreteSortSym (sortOfVariable code))
       [CONC_VAL "add"; CONC_VAL "mul"]
    8. list_holds_all (var_has_sort (sortOfVariable z))
       [sig_add; sig_mul]
    9. !t.
       (if assgn code t = CONC_VAL "add" then
         assgn z t = assgn sig_add t
        else
         (assgn code t = CONC_VAL "mul") /\
         (assgn z t = assgn sig_mul t))
   : goalstack
OK..
1 subgoal:
> val it =
   mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
   (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))
   ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. ~stateVariable z
    2. assgn code t = CONC_VAL "add"
    3. compatible_assgn assgn
    4. concreteVariable code
    5. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    6. list_holds_all (ofConcreteSortSym (sortOfVariable code))
       [CONC_VAL "add"; CONC_VAL "mul"]
    7. list_holds_all (var_has_sort (sortOfVariable z))
       [sig_add; sig_mul]
    8. !t.
       (if assgn code t = CONC_VAL "add" then
         assgn z t = assgn sig_add t
        else
         (assgn code t = CONC_VAL "mul") /\
         (assgn z t = assgn sig_mul t))
   : goalstack
OK..
1 subgoal:
> val it =
   mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
   (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))
   ------------------------------------
    0. compatible_assgn assgn /\ ~stateVariable sig_add /\
      (ranFunc mdg_add = sortOfVariable sig_add) /\
```

```
      ?vt.
       (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
       wellTypedTerm (TERM_FN mdg_add vt) /\
       !t.
         assgn sig_add t =
         intrp mdg_add (MDG_term_val intrp assgn vt t)
   1. ~stateVariable z
   2. assgn code t = CONC_VAL "add"
   3. compatible_assgn assgn
   4. concreteVariable code
   5. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
   6. list_holds_all (ofConcreteSortSym (sortOfVariable code))
       [CONC_VAL "add"; CONC_VAL "mul"]
   7. list_holds_all (var_has_sort (sortOfVariable z))
       [sig_add; sig_mul]
   8. !t.
       (if assgn code t = CONC_VAL "add" then
          assgn z t = assgn sig_add t
        else
          (assgn code t = CONC_VAL "mul") /\
          (assgn z t = assgn sig_mul t))
  : goalstack
OK..
1 subgoal:
> val it =
    compatible_assgn assgn /\ ~stateVariable sig_add /\
    (ranFunc mdg_add = sortOfVariable sig_add) /\
    (?vt.
      (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
      wellTypedTerm (TERM_FN mdg_add vt) /\
      !t.
        assgn sig_add t =
        intrp mdg_add (MDG_term_val intrp assgn vt t)) ==>
    mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
    (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))
    ------------------------------------
     0. ~stateVariable z
     1. assgn code t = CONC_VAL "add"
     2. compatible_assgn assgn
     3. concreteVariable code
     4. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
     5. list_holds_all (ofConcreteSortSym (sortOfVariable code))
         [CONC_VAL "add"; CONC_VAL "mul"]
     6. list_holds_all (var_has_sort (sortOfVariable z))
         [sig_add; sig_mul]
     7. !t.
```

```
        (if assgn code t = CONC_VAL "add" then
          assgn z t = assgn sig_add t
         else
          (assgn code t = CONC_VAL "mul") /\
          (assgn z t = assgn sig_mul t))
    : goalstack
OK..
1 subgoal:
> val it =
    assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))
    ------------------------------------
      0. ~stateVariable z
      1. assgn code t = CONC_VAL "add"
      2. compatible_assgn assgn
      3. concreteVariable code
      4. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
      5. list_holds_all (ofConcreteSortSym (sortOfVariable code))
           [CONC_VAL "add"; CONC_VAL "mul"]
      6. list_holds_all (var_has_sort (sortOfVariable z))
           [sig_add; sig_mul]
      7. !t.
           (if assgn code t = CONC_VAL "add" then
              assgn z t = assgn sig_add t
            else
             (assgn code t = CONC_VAL "mul") /\
             (assgn z t = assgn sig_mul t))
      8. ~stateVariable sig_add
      9. ranFunc mdg_add = sortOfVariable sig_add
      10. wellTypedTerm
            (TERM_FN mdg_add (TERM_PROD (TERM_VAR x) (TERM_VAR y)))
      11. !t'.
            assgn sig_add t' =
            intrp mdg_add (PROD_VAL (assgn x t') (assgn y t'))
      12. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    : goalstack
OK..
Meson search level: .......

Goal proved.
 [.............]
|- assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))

Goal proved.
 [........]
|- compatible_assgn assgn /\ ~stateVariable sig_add /\
   (ranFunc mdg_add = sortOfVariable sig_add) /\
```

```
 (?vt.
   (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
   wellTypedTerm (TERM_FN mdg_add vt) /\
   !t.
     assgn sig_add t =
     intrp mdg_add (MDG_term_val intrp assgn vt t)) ==>
 mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
 (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [.........]
|- mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
  (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [.........]
|- mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul ==>
  (assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [..........]
|- assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))

Remaining subgoals:
> val it =
   assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))
   ------------------------------------
     0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
     1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
     2. ~stateVariable z
     3. ~(assgn code t = CONC_VAL "add")
     4. ~(assgn code t = CONC_VAL "mul")
     5. compatible_assgn assgn
     6. concreteVariable code
     7. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
     8. list_holds_all (ofConcreteSortSym (sortOfVariable code))
          [CONC_VAL "add"; CONC_VAL "mul"]
     9. list_holds_all (var_has_sort (sortOfVariable z))
          [sig_add; sig_mul]
    10. !t.
         (if assgn code t = CONC_VAL "add" then
            assgn z t = assgn sig_add t
          else
            (assgn code t = CONC_VAL "mul") /\
            (assgn z t = assgn sig_mul t))
```

```
  assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t))
  ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
    1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    2. ~stateVariable z
    3. ~(assgn code t = CONC_VAL "add")
    4. assgn code t = CONC_VAL "mul"
    5. compatible_assgn assgn
    6. concreteVariable code
    7. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    8. list_holds_all (ofConcreteSortSym (sortOfVariable code))
         [CONC_VAL "add"; CONC_VAL "mul"]
    9. list_holds_all (var_has_sort (sortOfVariable z))
         [sig_add; sig_mul]
    10. !t.
          (if assgn code t = CONC_VAL "add" then
             assgn z t = assgn sig_add t
           else
             (assgn code t = CONC_VAL "mul") /\
             (assgn z t = assgn sig_mul t))
     : goalstack
OK..
1 subgoal:
> val it =
  mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
  (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))
  ------------------------------------
    0. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
    1. ~stateVariable z
    2. ~(assgn code t = CONC_VAL "add")
    3. assgn code t = CONC_VAL "mul"
    4. compatible_assgn assgn
    5. concreteVariable code
    6. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    7. list_holds_all (ofConcreteSortSym (sortOfVariable code))
         [CONC_VAL "add"; CONC_VAL "mul"]
    8. list_holds_all (var_has_sort (sortOfVariable z))
         [sig_add; sig_mul]
    9. !t.
         (if assgn code t = CONC_VAL "add" then
            assgn z t = assgn sig_add t
          else
            (assgn code t = CONC_VAL "mul") /\
            (assgn z t = assgn sig_mul t))
     : goalstack
OK..
```

```
1 subgoal:
> val it =
   mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
   (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))
   ------------------------------------
     0.  compatible_assgn assgn /\ ~stateVariable sig_mul /\
         (ranFunc mdg_mul = sortOfVariable sig_mul) /\
         ?vt.
           (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
           wellTypedTerm (TERM_FN mdg_mul vt) /\
           !t.
             assgn sig_mul t =
             intrp mdg_mul (MDG_term_val intrp assgn vt t)
     1.  ~stateVariable z
     2.  ~(assgn code t = CONC_VAL "add")
     3.  assgn code t = CONC_VAL "mul"
     4.  compatible_assgn assgn
     5.  concreteVariable code
     6.  list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
     7.  list_holds_all (ofConcreteSortSym (sortOfVariable code))
           [CONC_VAL "add"; CONC_VAL "mul"]
     8.  list_holds_all (var_has_sort (sortOfVariable z))
           [sig_add; sig_mul]
     9.  !t.
           (if assgn code t = CONC_VAL "add" then
              assgn z t = assgn sig_add t
            else
              (assgn code t = CONC_VAL "mul") /\
              (assgn z t = assgn sig_mul t))
  : goalstack
OK..
1 subgoal:
> val it =
   compatible_assgn assgn /\ ~stateVariable sig_mul /\
   (ranFunc mdg_mul = sortOfVariable sig_mul) /\
   (?vt.
     (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
     wellTypedTerm (TERM_FN mdg_mul vt) /\
     !t.
       assgn sig_mul t =
       intrp mdg_mul (MDG_term_val intrp assgn vt t)) ==>
   mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
   (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))
   ------------------------------------
     0.  ~stateVariable z
     1.  ~(assgn code t = CONC_VAL "add")
```

```
    2. assgn code t = CONC_VAL "mul"
    3. compatible_assgn assgn
    4. concreteVariable code
    5. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    6. list_holds_all (ofConcreteSortSym (sortOfVariable code))
         [CONC_VAL "add"; CONC_VAL "mul"]
    7. list_holds_all (var_has_sort (sortOfVariable z))
         [sig_add; sig_mul]
    8. !t.
         (if assgn code t = CONC_VAL "add" then
            assgn z t = assgn sig_add t
          else
            (assgn code t = CONC_VAL "mul") /\
            (assgn z t = assgn sig_mul t))
   : goalstack
OK..
1 subgoal:
> val it =
   assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t))
   ------------------------------------
    0. ~stateVariable z
    1. ~(assgn code t = CONC_VAL "add")
    2. assgn code t = CONC_VAL "mul"
    3. compatible_assgn assgn
    4. concreteVariable code
    5. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
    6. list_holds_all (ofConcreteSortSym (sortOfVariable code))
         [CONC_VAL "add"; CONC_VAL "mul"]
    7. list_holds_all (var_has_sort (sortOfVariable z))
         [sig_add; sig_mul]
    8. !t.
         (if assgn code t = CONC_VAL "add" then
            assgn z t = assgn sig_add t
          else
            (assgn code t = CONC_VAL "mul") /\
            (assgn z t = assgn sig_mul t))
    9. ~stateVariable sig_mul
    10. ranFunc mdg_mul = sortOfVariable sig_mul
    11. wellTypedTerm
          (TERM_FN mdg_mul (TERM_PROD (TERM_VAR x) (TERM_VAR y)))
    12. !t'.
          assgn sig_mul t' =
          intrp mdg_mul (PROD_VAL (assgn x t') (assgn y t'))
    13. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
   : goalstack
OK..
```

```
Meson search level: .......

Goal proved.
 [..............]
|- assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t))

Goal proved.
 [.........]
|- compatible_assgn assgn /\ ~stateVariable sig_mul /\
   (ranFunc mdg_mul = sortOfVariable sig_mul) /\
   (?vt.
     (vt = TERM_PROD (TERM_VAR x) (TERM_VAR y)) /\
     wellTypedTerm (TERM_FN mdg_mul vt) /\
     !t.
       assgn sig_mul t =
       intrp mdg_mul (MDG_term_val intrp assgn vt t)) ==>
   mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
   (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [..........]
|- mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
   (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [..........]
|- mdg_transform intrp assgn ([x; y],mdg_add) sig_add ==>
   (assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t)))

Goal proved.
 [...........]
|- assgn z t = intrp mdg_mul (PROD_VAL (assgn x t) (assgn y t))

Remaining subgoals:
> val it =
    assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))
    ------------------------------------
      0. mdg_transform intrp assgn ([x; y],mdg_add) sig_add
      1. mdg_transform intrp assgn ([x; y],mdg_mul) sig_mul
      2. ~stateVariable z
      3. ~(assgn code t = CONC_VAL "add")
      4. ~(assgn code t = CONC_VAL "mul")
      5. compatible_assgn assgn
      6. concreteVariable code
      7. list_all_distinct [CONC_VAL "add"; CONC_VAL "mul"]
      8. list_holds_all (ofConcreteSortSym (sortOfVariable code))
```

```
        [CONC_VAL "add"; CONC_VAL "mul"]
    9.  list_holds_all (var_has_sort (sortOfVariable z))
        [sig_add; sig_mul]
    10.  !t.
         (if assgn code t = CONC_VAL "add" then
            assgn z t = assgn sig_add t
          else
            (assgn code t = CONC_VAL "mul") /\
            (assgn z t = assgn sig_mul t))
   : goalstack
OK..
Meson search level: ...

Goal proved.
 [...........]
|- assgn z t = intrp mdg_add (PROD_VAL (assgn x t) (assgn y t))

Goal proved.
 [...]
|- mdg_mux assgn
     (code,
      [(string_to_val "add",sig_add); (string_to_val "mul",sig_mul)])
     z ==>
   outTable intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t

Goal proved.
 [....]
|- outTable intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t

Goal proved.
 [...]
|- table intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y) t

Goal proved.
 [...]
|- TABLE intrp assgn [vtt code] z [[conctv "add"]; [conctv "mul"]]
     [add_t x y; mul_t x y] (add_t x y)
> val it =
    Initial goal proved.
    |- !intrp assgn code x y z q.
         ALU_str intrp assgn (code,x,y) z ==>
         ALU_beh intrp assgn (code,x,y) z : goalstack
> val alu_rws =
```

```
  [|- !assgn a b cin s cout.
       RCA4_str assgn (a,b,cin) (s,cout) =
       ?a0 a1 a2 a3 b0 b1 b2 b3 c0 c1 c2 s0 s1 s2 s3.
         wires assgn a [a0; a1; a2; a3] /\
         wires assgn b [b0; b1; b2; b3] /\
         wires assgn s [s0; s1; s2; s3] /\
         FA_str assgn (a0,b0,cin) (s0,c0) /\
         FA_str assgn (a1,b1,c0) (s1,c1) /\
         FA_str assgn (a2,b2,c1) (s2,c2) /\
         FA_str assgn (a3,b3,c2) (s3,cout),
   |- !assgn a b cin s cout.
       RCA4_beh assgn (a,b,cin) (s,cout) =
       !t.
         ?sw.
           (sw =
            NBWORD 5
              (BNVAL (val_to_word (assgn a t)) +
               BNVAL (val_to_word (assgn b t)))) /\
           (MDG_value_to_bool (assgn cout t) = MSB sw) /\
           (val_to_word (assgn s t) = WSEG 4 1 sw)] : thm list
<<HOL message: inventing new type variable names: 'a>>
> val it =
    Proof manager status: 2 proofs.
    2. Completed:
         |- !intrp assgn code x y z q.
              ALU_str intrp assgn (code,x,y) z ==>
              ALU_beh intrp assgn (code,x,y) z
    1. Incomplete:
         Initial goal:
         !assgn a b cin s cout.
           RCA4_str assgn (a,b,cin) (s,cout) ==>
           RCA4_beh assgn (a,b,cin) (s,cout)


    : proofs
OK..
2 subgoals:
> val it =
    val_to_word (assgn s t) =
    WSEG 4 1
      (NBWORD 5
         (BNVAL (val_to_word (assgn a t)) +
          BNVAL (val_to_word (assgn b t))))
    ------------------------------------
      0.  wires assgn a [a0; a1; a2; a3]
      1.  wires assgn b [b0; b1; b2; b3]
```

```
   2. wires assgn s [s0; s1; s2; s3]
   3. FA_str assgn (a0,b0,cin) (s0,c0)
   4. FA_str assgn (a1,b1,c0) (s1,c1)
   5. FA_str assgn (a2,b2,c1) (s2,c2)
   6. FA_str assgn (a3,b3,c2) (s3,cout)

  MDG_value_to_bool (assgn cout t) =
  MSB
   (NBWORD 5
     (BNVAL (val_to_word (assgn a t)) +
      BNVAL (val_to_word (assgn b t))))
  ------------------------------------
   0. wires assgn a [a0; a1; a2; a3]
   1. wires assgn b [b0; b1; b2; b3]
   2. wires assgn s [s0; s1; s2; s3]
   3. FA_str assgn (a0,b0,cin) (s0,c0)
   4. FA_str assgn (a1,b1,c0) (s1,c1)
   5. FA_str assgn (a2,b2,c1) (s2,c2)
   6. FA_str assgn (a3,b3,c2) (s3,cout)
   : goalstack
[closing file "aluScript.sml"]
> val it = () : unit
-
```