A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs

Subhashini Balakrishnan

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

November 1999

Abstract

A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs

Embedded systems are finding widespread application including communication systems, factory automation, graphics and imaging systems, medical equipment and even household appliances. With the increasing emergence of mixed hardware/software systems, it is important to ensure the correctness of such a system formally, particularly for real-time and safety critical applications. In this thesis, a hierarchical approach to modeling and formally verifying a complete embedded system at higher levels of abstraction, using Multiway Decision Graphs (MDGs), is proposed. The approach is demonstrated on the embedded software for a mouse controller application on a commercial microcontroller (PIC 16C71) from Microchip Technologies Inc..

The embedded system is modeled at different levels of the design hierarchy i.e., the microcontroller RT level, the microcontroller Instruction Set Architecture (ISA), the embedded software assembly code level and the embedded software flowchart specification. The correctness of the system hardware platform in implementing its intended architecture is established by formally verifying the equivalence between the RTL hardware and the ISA, using the MDG sequential equivalence checking tool. The next step is taken to verify the particular application embedded in the system by checking the equivalence between the assembly code and its intended behavior, specified as a flowchart.

Further verification is done on the models through the property checking procedure provided by the MDG tools. Liveness properties are also checked using the newly developed MDG model checking procedure.

Inconsistencies in the assembly code with respect to the specification, as published in the application notes of the manufacturer, were uncovered through the verification experiments. Given the relatively small CPU time and memory consumption achieved in the experiments, the verification approach that is adopted was able to verify a whole embedded system in an automated environment.

Acknowledgments

As a student very much interested in design of digital systems, little was known about formal verification before my graduate studies. My supervisor Prof. Sofiène Tahar sufficiently motivated me to take up graduate work in this field. Without the enthusiasm and guidance from Prof. Sofiène Tahar and the MDG group at University of Montreal, I wouldn't have been able to complete my thesis work.

I would like to express my special gratitude to Dr. Otmane Ait-Mohamed, Nortel Networks for showing keen interest in my work and for taking his valuable time to address my concerns. I also thank Dr. Ying Xu, Nortel Networks, for her time and help in my MDG model checking experiments. I also enjoyed working with Ali Abbas Mir on the SMV model checking experiments.

I should mention my friends, Vijay Pisini, Krishnan, Srinivasan, Kumuthini, Rachit and Seetharaman, who have made my life in school a memorable one. Mr. Tharmagaran and family were always a source of invaluable support and a sanctuary whenever I needed one.

Above all, next to the love of the Almighty, nothing can replace the love of my grandparents, my parents, my brothers Giridharan, Jeyendran and family, Karthikeyan and family, and my husband. It is amazing the way they bring out the best in me. They make my life worth living.

Dedication

To my wonderful grandparents, Mr. & Mrs. Panchapagesan, Mr. & Mrs. Ramasamy, and my darling Geerthan

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Formal Verification

Hardware and software designs are rapidly increasing in complexity. Traditionally, testing and simulation are used to check the design correctness. Simulation catches some problems, but not exhaustively. The increasing concurrence and complexity of designs exacerbates this problem: detecting every bug resulting from the complex interaction of concurrent events by simulation becomes highly improbable (or prohibitively time consuming). Testing and simulation are inadequate to certify that a system behaves correctly. High costs (time, money, security and possibly lives) are incurred because a system is typically delivered with design errors, and hence must go through several design iterations. Improved debugging tools and methodologies are critical to avoid the expenses and delays resulting from discovering bugs late in the design phase.

To accelerate the design and assure the correctness of complex systems, a hierarchical design approach, as shown in Figure 1.1 is usually adopted [30]. The designer first manually derives the requirements of the system as the system behavioral specification. This specification is then refined manually or using CAD tools into more detailed descriptions such as register-transfer (RT), logic and mask level descriptions. As the late detection of design errors is largely responsible for unexpected delays in realizing the hardware design, it is extremely important to ensure correctness in each design step. With

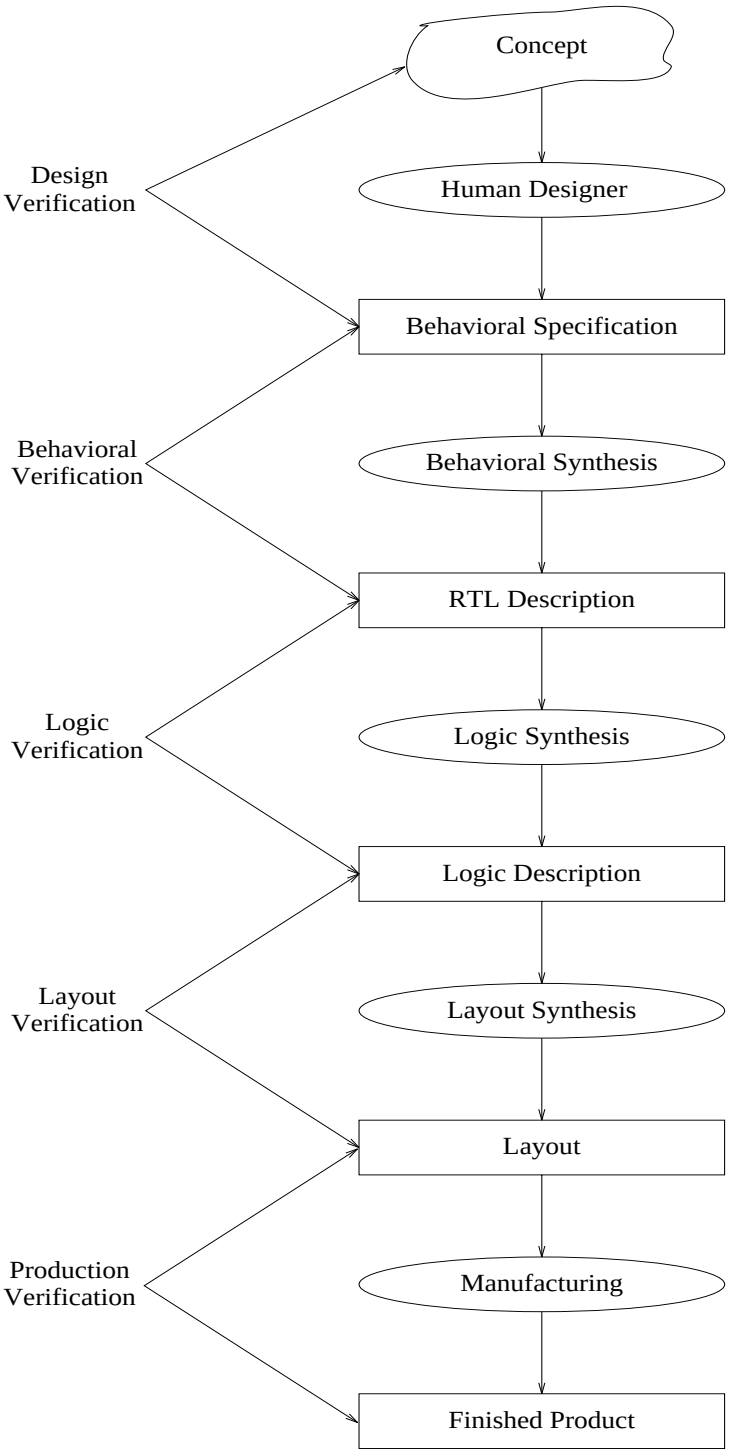


Figure 1.1: A Hierarchical Design Methodology

correct-by-construction design style, automatic tools, such as behavioral and logic synthesis techniques can be used to ensure behavioral and gate level design correctness. However, the refinement process from high-level specification to synthesizable design usually requires manual fine tuning to achieve high performance. More progress is needed to automate the design process at higher levels in order to produce designs of the same quality as is achievable by hand. It is thus essential that the specification (or behavior) and the intermediate design stages be verified for consistency and correctness with respect to some user-specified properties or a previous level of the specification, thus making post-design verification essential.

This situation has prompted interest in verification techniques. Formal methods have long been developed and advocated within the computing science research community as providing sound mathematical foundation for the specification, implementation and verification of computer systems. These methods exploit representations with formally defined semantics in order to describe abstractly (independent of details of implementation) the desired functional behavior of a system [4]. Such formalization methods provide precise and unambiguous system specifications which can be checked for completeness and internal logical consistency. The mathematical nature of these specifications enable reasoning about consistency (i.e., whether the system dynamics is consistent with system's static properties) and the deduction of consequences of the specification. These can be checked against the user's expectations and used to generate tests for the system implementation.

Specifications in an executable formal language allow direct simulations (animations) of system behavior, giving early feedback to be compared with user requirements before full system development is begun. Equally important in the system development process, a formal specification is a yardstick against which to verify implementations or

implementation steps through mathematical proof of the equivalence of abstract and concrete representations of system operations or data structures [36]. A formally based development methodology requires in effect that a mathematical theory of the desired system be created, documented and analysed. This foundation activity entails a greater proportion of time and effort being invested in the initial pre-design phases of system development than is now commonly the case.

Thanks to the rigorous discipline imposed by these methods, system development phases are rendered less error-prone, more systematic and amenable to computer assistance, and hence higher quality products achieved. Thus, formal verification is proposed as a method to help certify hardware and software, and consequently, to increase confidence in new designs. Formally verifying designs may be cost effective in “safety critical” applications, for systems in high volume or remotely placed, and for systems that will go through frequent redesign because of changes in technology. Recently, formal verification has been considered as a powerful complementary approach to simulation and has made exciting progress [34].

1.2 Formal Verification Methodologies

Various formal verification methodologies have sprung up in the recent past. They could be classified based on the following:

- Proof method:
  - Theorem proving
  - Model checking
  - Machine equivalence

- Language containment/equivalence
- Trace conformance

• User interaction:

- Interactive verification
- Automated verification

• Class of circuits we wish to verify:

- Combinational/Sequential
- Synchronous/Asynchronous
- Pipelined hardware
- Parameterized hardware

1.2.1 Theorem proving

With theorem proving, an implementation and its specification are usually expressed as first-order or higher-order logic formulas. Their relationship, stated as equivalence or implication, is regarded as a theorem to be proven within the logic system, using axioms and inference rules. Thus, theorem proving is a powerful verification technique. It can provide a unifying framework for various verification tasks at different hierarchical levels. However, the task of proving complex theorems needs expertise. A theorem prover or proof checker is a tool developed to partially automate the proof process or to check a manual proof. Theorem proving systems are being widely used both in the hardware and software verification, on an industrial scale. Some of the well-known ones are HOL (Higher-Order Logic) [33], PVS (Prototype Verification System) [49], Nqthm (a Boyer-Moore theorem prover) [5] and ACL2 [39].

1.2.2 FSM-based verification

In FSM-based verification, synchronous sequential designs are modeled as finite state machines. These models are represented using data structures known as Binary Decision Diagrams (BDDs). The basic method of verification is based on automated state enumeration of the FSM, called *reachability analysis* [23, 59, 60]. Both equivalence and property checking can be performed using this technique. Equivalence checking verifies that an implementation has the same outputs as that of the specification, for all input sequences, wherein both the implementation and the specification are modeled as FSMs. Property checking verifies the validity of a specification, expressed as a set of properties, on an implementation, modeled as an FSM. A well-known approach for property verification is Computational Tree Logic (CTL) model checking [44]. A specification for model checking is a collection of properties, expressed in CTL which can concisely capture temporal relationships between states. The model represents the model of the system which is to be verified.

The major advantage of the FSM-based verification is automation, apart from rendering easier formalization of issues like concurrence, fairness etc. But, the two serious drawbacks are state explosion and boolean representation. The use of Bryant's ROBDDs [8] reduces the complexity to linear, with respect to the width of the datapath, for certain kinds of circuits. This significantly enlarged the useful domain of the FSM-based verification. A number of tools have been developed, the two well-known among them are the SMV (Symbolic Model Verifier) [44] and the VIS (Verification Interacting with Synthesis) [6].

The limitations of the FSM-based methods have been attacked in two main directions: Problem reduction and representation of hardware at different abstraction levels.

Problem reduction techniques such as data abstraction, homomorphic reductions, data-independent systems, or restating a verification problem in terms of the controller alone are all restricted to the particular problem at hand and the equivalence of the original problem to the reduced or restricted or restated ones is not always obvious and is not verified mechanically. This requires a user's ingenuity for each particular problem.

Recently a number of ROBDD extensions such as BMDs [9], HDDs [16] or K*BMDs [25] have been developed to represent arithmetic functions more compactly than ROBDDs. An improvement is the EOBDDs [41] that can have leaf nodes labelled by terms containing abstract sorts. MDGs, the successor of EOBDDs, allow the labelling of edges to be first order terms and non-terminal nodes to be abstract variables. ROBDDs, MDDs and MTBDDs are special cases of MDGs and can be turned into MDGs by transforming them from graphs representing functions into graphs representing relations.

In this thesis, MDGs are used to model and verify embedded systems, automatically in a hierarchical manner. Chapter 2 is devoted to giving details on MDGs and the MDG verification system.

1.3 Embedded systems

Advances in VLSI and synthesis technology have made it flexible to construct powerful programmable components (microprocessors) as well as complex specialized components. Today, electronic products consist of a mixture of hardware and software components. An embedded system is regarded as a product which contains a microprocessor programmed to carry out some control functions but which is not itself a computer [27]. An embedded system encompasses a broad class of systems, ranging, in principle, from a simple

microprocessor based apparata to complex systems controlling large plants, aircrafts and the like.

In general [40]:

(1) An embedded system is an electronic system embedded within a given plant or external process. The external process comprises both a physical system (usually consisting of different subsystems) and also humans performing some supervising or parameter setting tasks.

(2) Most embedded systems must fulfill stringent reliability requirements, usually detailed according to a set of functions to be performed.

1.4 Related Work

In the recent past significant success was attained in verifying microprocessor hardware designs using various approaches including theorem provers, verification using meta languages, functional approaches and decision diagram based approaches.

Gordon [32] first verified a simple computer using the LCF_LSM system [31], which was an early example of how formal proof and mechanical proof-generation could be used to reason about the design of a microprocessor. Hunt [35] proceeded to verify the microprocessor FM8501 using Boyer-Moore theorem prover [5]. Hunt was the first to consider the implementation of a handshaking protocol in a microprocessor system. Cohn [18, 19, 20] verified the commercial microprocessor VIPER using HOL [33]. Cohn came up with two level of proofs, the first level dealing with the flow of control, and the second level dealing with the block level description. The VIPER project shows how the design of a microprocessor can be subjected to formal analysis in a series of decreasingly abstract

levels. Joyce verified the Tamarack-3 microprocessor [38] using HOL. The verification is done at a high level of abstraction that he did not even mention the width of the datapath. He gave a generic specification of a simplified multi-layered Tamarack stack, categorized into a compiler and a microprocessor, and described how to link the compiler to the microprocessor. Windley [56] also proposed a general methodology for verifying generic interpreters of micorprogrammed processors, using HOL. Srivas and Bickford [51] verified the pipelined microprocessor MiniCayuga using the Clio theorem proving system. Tahar and Kumar [53] proposed a general methodology for verifying pipelined RISC processors using HOL. Srivas and Miller [52] reported the verification of a modern complex commercial processor, AAMP5 using PVS [49].

Recently, a number of automatic verification methods have been explored for verifying microprocessor designs. Burch and Dill's [12, 37] validity checking algorithm is an efficient approach for instruction set processor verification. A logic expression representing the correctness statement is generated using symbolic simulation. The validity checking algorithm is then used to verify if the expression is valid. With carefully chosen heuristics to avoid exponential case splitting, the authors verified a subset of the RISC pipeline processor DLX [12] and a protocol processor [37]. Galter [29] presented a similar approach for the verification of processors. Two ITE-expressions (If-Then-Else) which represent the functions of the specification and the implementation are derived using symbolic execution. They are then compared for syntactic equivalence. A technique called *IF-algebra* was developed to simplify the exponentially growing IF expressions. The Tamarack-3 microprocessor benchmark was verified using this method. Verification

methodologies combining symbolic simulation and theorem proving were also explored by Barringer [3] and Cyrluk and Narendran [24].

As more and more processors are being specialized for embedded applications, there arose an imperative need to focus on verifying the software embedded in the processors. Thiry and Claesen [54] suggested a methodology for formally verifying an embedded software [45] running on a microcontroller [46], using the SMV tool [44]. Their intention was to model the machine architecture as an instruction interpreter and the assembly code as a finite state machine. They presented a model of the execution of the embedded assembly language software on the microcontroller hardware. The model is represented at the boolean level. They used the flowchart specification of the embedded software to derive properties of the software routine, and represented them using CTL temporal logic [26]. They verified the properties on the software model using the SMV tool. The SMV uses the ROBDD symbolic model checking algorithm [11] to find out whether the CTL specifications are satisfied on the model. More recently, Brock and Hunt [7] specified and verified programs for the Motorola Complex Arithmetic Digital Signal Processor (CAP) using ACL-2 theorem-proving system [39]. They completely specified the CAP super-scalar processor in a high level behavioral model and mechanically verified a simple FIR filter and a high-speed searching algorithm, represented as machine code, using ACL-2.

1.5 Scope of Thesis

Interest in hardware/software codesign [10] has been on the rise for the past couple of years, and this interest has been manifesting itself in the emergence of exotic tools to facilitate the design of entire systems. With the increasing application of mixed hardware/

software systems in embedded computers and safety critical systems, there is need to produce high integrity systems that are correct in all situations. Several authors have demonstrated the infeasibility of showing that such systems meet ultra-high reliability requirements through testing alone [13, 42]. Although completely reliable systems cannot be guaranteed, the use of formal methods is an alternative approach that systematically analyses all cases in a design and specification.

The work of Thiry and Claesen [54] and Brock and Hunt [7] provided an inspiration to expand the scope of formal verification into embedded system verification. While a symbolic model checker is restricted to representation at the boolean level, a theorem prover is restricted to users with a lot of expertise and experience. Thus, the motivation behind this work is the search for a methodology that could handle the modeling and verification of a whole embedded system at various levels of abstraction, that could enable the verification to be integrated into the design process. The verification tools based on MDGs presented a way for experimenting with such an approach. Several successful hardware verification results have been reported using the MDG verification system [13]. The preliminary results obtained in [1, 2] provided an encouragement behind this work.

In this thesis, an application of formal methods to verify embedded systems in a hierarchical manner, using Abstract State Machines (ASMs) [23], based on Multiway Decision Graphs (MDGs) [23] is proposed. The approach is demonstrated on an embedded software for a serial mouse controller application [45] programmed on the microcontroller PIC16C71, commercialized by Microchip Technology Inc., [46]. It illustrates the ability to carry out equivalence checking, in addition to checking properties, using ASMs. Thus, it

paves a way for automatic verification of a complete embedded system at higher levels of abstraction.

The rest of this thesis is organized as follows: Chapter 2 is a brief introduction to Multiway Decision Graphs and its related verification techniques. Chapters 3 and 4 illustrate, through simple examples, the hierarchical approach to modeling and verification of an embedded system using MDGs. Chapter 5 gives a description of an embedded system case study and the application of the hierarchical modeling and verification on the target system. The model checking experiments performed on the target system are also reported, along with a comparison study of the results obtained using MDG model checking with that of SMV model checking. The conclusions and ideas on further work are presented in Chapter 6.

Chapter 2

Multiway Decision Graphs

Multiway Decision Graphs (MDGs) have been proposed recently [23] to represent circuits with datapath. The MDG tool combines the advantages of representing a circuit at higher abstract levels as is possible in a theorem prover, and of the automation offered by ROBDD based tools. MDGs, a new class of decision graphs, comprises, but is much broader than the class of ROBDDs.

2.1 Multiway Decision Graphs

The formal system underlying MDGs is a subset of many-sorted first order logic, augmented with a distinction between *abstract* and *concrete* sorts. Concrete sorts have *enumerations* while abstract sorts do not. The enumeration of a concrete sort α is a set of distinct constants of sort α. The constants occurring in enumerations are referred to as *individual constants*, and other constants as *generic constants* and could be viewed as 0-ary function symbols. The distinction between abstract and concrete sorts lead to a distinction between three kinds of function symbols. Let *f* be a function symbol of type $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort, then *f* is an *abstract function symbol*. If all the $\alpha_1 ... \alpha_{n+1}$ are concrete, then *f* is a *concrete function symbol*. If α_{n+1} is concrete while at least one of the $\alpha_1 ... \alpha_n$ is abstract, then *f* is referred to as a *cross-operator*. Concrete function symbols must have explicit definition; they can be eliminated and do not appear in MDGs. Abstract function symbols and cross-operators are *uninterpreted*.

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of concrete sort with its edge labels being the individual constants in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labeled by abstract terms of the same sort; or it can be a cross-term (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as **T**, which means all paths in an MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs can also represent sets of states.

2.2 Modeling Hardware with MDGs

Using MDGs a data value can be represented by a single variable of abstract sort, rather than by concrete Boolean variables. Variables of abstract sort are used to denote data signals and *uninterpreted function symbols* to denote data operations. Cross-operators (a special case of uninterpreted functions) are useful for modeling feedback from datapath to the control circuitry. They are thus much more compact than ROBDDs for designs containing datapath, and sequential circuits can be verified independently of the width of the datapath.

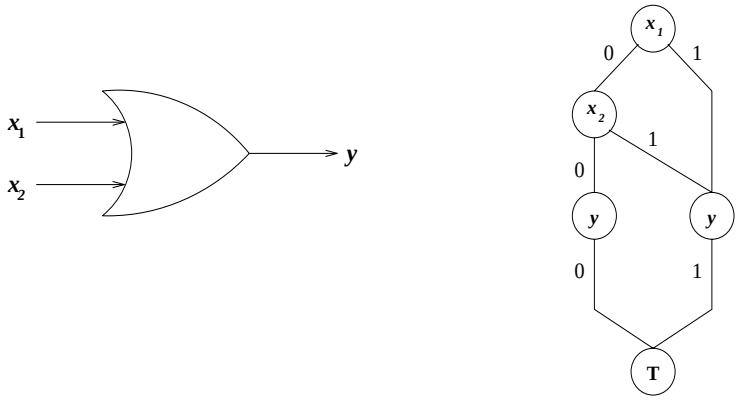


Figure 2.1: MDG for an OR gate

Fig. 2.1 shows an OR gate and its MDG representation, for a particular ordering of the variables. Boolean MDGs are essentially the same as ROBDDs. In the MDG system, abstract descriptions of state machines, called *Abstract State Machines* (ASMs) [23] are used to model the systems. ASMs are a new way of describing state machines. They admit non-finite state machines as models in addition to their intended finite interpretations. An ASM is obtained by letting some data input, state or output variables of a finite state machine (FSM) be of abstract sort, and the datapath operations be uninterpreted function symbols. Fig. 2.2 shows a tabular description of a simple ASM, with its MDG representation, where *x* is a Boolean input, *a* is an abstract state variable and *a'* is its next state variable. It performs *inc* operation when *x* = 1, where *inc* is an uninterpreted function symbol.

x	***a'***
0	a
1	inc(a)

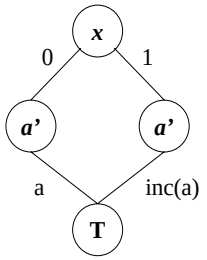


Figure 2.2: MDG for a simple ASM

In analogy to ROBDDs, which are used to represent sets of states and transition/output relations for FSMs, MDGs are used to compactly encode sets of (abstract) states and transition/output relations for ASMs. Thus the implicit enumeration technique [55] is lifted from the Boolean level to the abstract level, and refer to it as *implicit abstract enumeration* [22]. This makes it possible to verify a circuit at the register transfer (RT) level without getting bogged down with the details of a gate level implementation. Thereby, the use of

ASMs raises the level of abstraction of automated verification methods to approach those of interactive theorem proving methods, without sacrificing automation.

2.3 MDG-based Verification

Like ROBDDs, MDGs must be *reduced* and *ordered*. They obey a set of well-formedness conditions, which turns MDGs into a canonical representation, which is used by the combinational equivalence checking procedure of the MDG tools (Section 2.3.1). This is, unfortunately, not of much use in the *reachability analysis* procedure, because the descriptions of the sets of states involves an implicit existential quantification over abstract variables which removes the canonicity property.

Algorithms for computing *disjunction*, *relational product* (*conjunction* followed by existential quantification [23]), *pruning-by-subsumtion* (*PbyS*, for test of set inclusion [23]) and *reachability analysis* (using implicit abstract enumeration) have been implemented in the MDG software package [14]. Except for *PbyS*, the operations are a generalization of first-order terms of algorithms on ROBDD, with some restrictions on the appearance of abstract variables in the arguments. Since in the underlying logic of MDG there is no complement of expression involving equality over abstract terms, *PbyS* approximates the relative complement between two formulas P and Q, by removing from P those MDG paths (conjuncts) that are subsumed by some paths in Q. Namely, if R=*PbyS*(P, Q), then $\models R \vee (\exists U)\ Q \Leftrightarrow P \vee (\exists U)\ Q$ [59].

In the *reachability analysis* procedure, starting from the set of initial states, the set of states reached in one transition is computed by the relational product operation. The frontier set of states is obtained by removing the already visited states from the set of newly

reached states using the pruning-by-subsumtion (*PbyS*) operation. If the frontier set of states is empty, then the *reachability analysis* procedure terminates, since there are no more unexplored states. Otherwise the newly reached states are merged (using *disjunction*) with the already visited states and the procedure continues the next iteration, with the states in the frontier set as the set of initial states.

In addition to the logic operations, a facility to carry out simple rewriting of terms that appear in the MDGs is also included. This allows us to provide a partial interpretation to (some) of the uninterpreted function symbols. For example, if *zero* is an abstract generic constant of sort *wordn* and *eqz*(*x*) a cross-operator of type [*wordn*⟶*bool*], then we could provide a partial interpretation of *eqz* using the rewrite rule *eqz*(*zero*)⟶*1,* indicating that *equal-to-zero* is 1 when the argument is *zero* (but not revealing anything about the other values). User selected rewrite rules are applied anytime a new term is formed during MDG operations. In general, rewriting simplifies MDGs and helps remove false negatives during safety property checking, thus likely avoiding non-termination of the *reachability analysis* procedure for designs that depend on interpretation of operators for correct operation. A detailed description of the operations and algorithms can be found in [59]; some possible solutions to the non-termination problem are addressed in [14].

MDGs are used as the underlying representation for a set of hardware verification tools, providing both validity checking and verification based on state-space exploration [23]. The MDG tools package the basic MDG operators and verification procedures. The operators are disjunction, pruning-by-subsumption, and term-rewriting. The following are the verification procedures provided in the MDG software package:

2.3.1 Combinational Verification

The combinational verification provides equivalence checking and safety property checking of two combinational circuits. The MDGs representing the input-output relation of each circuit are computed using the relational product of the MDGs of the components of the circuits. Then, taking advantage of the canonicity of MDGs, it is verified whether the two MDG graphs are isomorphic.

2.3.2 Safety Property Checking

A given safety property, a logic expression, is expressed as an invariant condition. An invariant condition can be specified by a combinational circuit whose output signals are named by the variables that occur in the condition. An MDG representing the invariant condition is obtained from the MDG representing the functionality of the combinational circuit by existentially quantifying the concrete inputs. The variables representing abstract inputs are left in the graph as implicitly quantified secondary variables. The state space of the given circuit (modeled as an ASM) is explored in each state using symbolic *reachability analysis*. It is verified that the specified property is satisfied (i.e., it is invariant over the reachable state space) by the given circuit.

2.3.3 Sequential Verification

Sequential verification provides equivalence checking of two component state machines. The transition relation of the two ASMs is represented by an MDG, computed by the relational product algorithm from the MDGs of the components, which are themselves abstract machines. In other words, the relational product computes the (synchronous) product machine of the component ASMs. The behavioral equivalence of

two sequential circuits, modeled as ASMs, is verified by checking whether the circuits produce the same sequence of outputs for every sequence of inputs. This is achieved by forming the product circuit consisting of the two circuits, feeding the same inputs to both of them, and verifying an invariant asserting the equality of the corresponding outputs in all reachable states.

2.3.4 Model Checking

Model checking feature has been recently developed [58] and incorporated into the existing MDG system. This provides both safety and liveness property checking using the implicit abstract enumeration of an ASM [57]. The properties are represented in a first-order linear time temporal logic, called L_{MDG}. The ASM model of the L_{MDG} formula is constructed, along with a simplified invariant. The ASM of the L_{MDG} is composed with the original model and the simplified invariant is checked on the composite machine, using the implicit abstract enumeration of an ASM [57].

2.3.5 Counterexample Generation

When invariant checking fails, the MDG tools generate a counterexample to help tracing the source of the error. A counterexample consists of a list of assumptions, input and state values in each clock cycle, which provides a trace leading from the initial state to the faulty behavior. At present, the counterexample generation feature is not provided in the MDG model checking algorithm.

The MDG system uses MDG-HDL [60] as the front-end description language, which allows the use of abstract variables and uninterpreted function symbols. MDG-HDL supports structural descriptions, behavioral ASM descriptions, or a mixture of structural

and behavioral descriptions. A structural description is usually a (hierarchical) network of components (modules) connected by signals.

The MDG-HDL comes with a large library of predefined, commonly used, basic components (such as logic gates, multiplexers, registers, bus drivers, ROMs, etc.) [60]. A behavioral description is given by high-level constructs as ITE (If-Then-Else) formulas, CASE formulas or tabular representations. The tabular constructor is similar to a truth table but allows first-order terms in rows. The MDG-HDL description is then compiled into the ASM model in internal MDG data structures [60]. The MDG tools run on a Prolog platform. Interested readers are referred to [14, 23, 57, 58, 59, 60] for more details on the MDG algorithms and tools.

Chapter 3

Hierarchical Modeling of Embedded Systems with MDGs

3.1 Introduction

A *specification* and an *implementation* are two descriptions of a system, be it hardware, software, or both, where the specification is a more abstract view of an implementation. A specification could be a behavioral description, which again could be complete or partial. In the latter case, the specification would be a set of properties. In the MDG system, an ASM can be used to describe a specification or an implementation. In our application, each state in the ASM characterizes the contents of the microcontroller registers. The statements of a specification or the instructions in an assembly code control the transition from one state to another.

An embedded system comprises both the hardware and the embedded application software. A hierarchical approach to modeling an embedded system is proposed at different levels of the design hierarchy, thus enabling the verification to be applied at various stages during the design process. The system is modeled in a hierarchical approach at four distinct levels namely, the embedded system hardware architecture at the RT level, the Instruction set architecture, the embedded software behavioral specification, and its assembly language implementation.

3.2 Microcontroller RTL Architecture

A common general purpose microcontroller comprises the following basic components: program memory, instruction register, instruction decoder, register file, ALU, working register, a system bus, and a program counter. One of the prime advantage in using MDGs is the ability to handle abstract descriptions. This avoids all the cumbersome procedure of defining each bit of a register. Rather, a register can be viewed as an abstract variable. Thus, an 8-bit general purpose register of the microcontroller can be modeled as a variable of abstract sort *worda8* instead of a concrete sort with enumeration {0,...., 255}. A program counter (say, 10-bit wide) can be modeled as a variable of abstract sort *worda10*. Another advantage in using MDGs is the ability to represent uninterpreted functions. This enables the program memory, register file, and the ALU to be viewed as black boxes. For instance, the uninterpreted function symbol *fetch* is used to model the instruction fetch from the program memory. A register file accesses are described in terms of functions like *read* and *write*, modeled as uninterpreted function symbols. The ALU functions are expressed using uninterpreted function symbols e.g., *add*, *sub*, *inc*, *or* etc. In MDG-HDL, the system bus can be modeled using the basic component *drivers*.

3.3 Microcontroller IS Architecture

The instruction set of a microcontroller typically includes mathematical operations, logical operations, transfer operations, control operations and no-operation. Using MDGs the assembly instructions can be modeled as predicates. Predicates can be described using the MDG-HDL basic library function, *transform*. The operations (e.g., mathematical, logical) can be modeled using uninterpreted functions (e.g., *add*, *inc*, *or*, *move*, *goto*),

applied to the arguments of the instructions (predicates). This is illustrated on the following two instructions: the inclusive-OR instruction (`OR R W`) between a register (`R`) and a working register (`W`), and the increment instruction (`INCR W`) of the working register (`W`).

Assembly Instruction: `OR R W`

MDG-HDL Model:

Definitions:
```
var(W, worda8)
var(R, worda8)
function(or, inputs[worda8, worda8], output[worda8])
```

Instruction:
```
transform(inputs([W, R]), function(or), output(W))
```

Assembly Instruction: `INCR W`

MDG-HDL Model:

Definitions:
```
var(W, worda8)
function(inc, input[worda8], output[worda8])
```

Instruction:
```
transform(inputs([W]), function(inc), output(W))
```

The instruction fetch is modeled using the uninterpreted function symbol *fetch*. Decoding specific bits of an instruction can be modeled in a similar way, e.g., the uninterpreted function *get_opcode* models the decoding of the operation from the instruction, and *get_src_op* and *get_dest_op* models the decoding of the source and destination operands respectively from the instruction.

3.4 Embedded Software

The embedded software models are represented as ASMs. In MDG-HDL, an ASM is described mainly in terms of a tabular representation. Each row in a table represents a CASE statement. Each column consists of an input condition and the last column gives either an output or a transition relation of a state variable. The default value for a CASE statement can be shown in an optional row [60]. In the following, models for the behavior of an example embedded software and its implementation in assembly language code, using MDGs, are presented.

3.4.1 Embedded Software Behavior

One common way of describing the intended behavior is by using algorithmic flowcharts. Hence, we consider the flowchart of the embedded software program as the behavioral specification. An example flowchart, its ASM model and its description in MDG-HDL is shown in Figure 3.1. The ASM model consists of three states, which are labeled “S0” to “S2”. Each state implements one step in the flowchart. A step may consist of many operations that could be done concurrently. In the MDG-HDL model the registers *R1* and *R2* are of abstract sort *worda8*, and *b1* of abstract sort *wordc3*. The model uses the abstract functions *decr*, *setb* and *resetb* to decrement, set and clear a register respectively, and the cross function *testbitforzero* to test a particular bit of a register for zero condition. They are defined with following types[1]:

decr: [*worda8* $\rightarrow$ *worda8*]

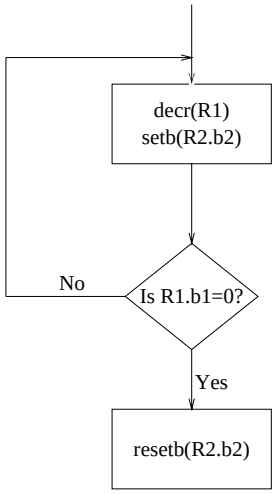
setb: [*worda8*, *worda3* $\rightarrow$ *worda8*]

1. The notation *f*: [$\alpha \rightarrow \beta$] implies that the function *f* has argument of sort α and range of sort β.

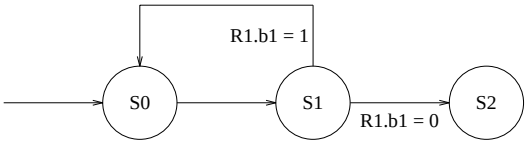
resetb: [*worda8, worda3* → *worda8*]

testbitforzero: [*worda8, worda3* → *bool*]

Flowchart:



ASM model:



MDG-HDL model:

```
initialize s = 0
case s = 0:   R1 = decr(R1)
              R2 = setb(R2,b2)
              next(s) = 1
case s = 1:   if testbitforzero(R1,b1) = 0
                 then next(s) = 2
                 else next(s) = 0
case s = 2:   R2 = resetb(R2,b2)
```

Figure 3.1: An example flowchart and its ASM model

While, each step in the flowchart must be executed in succession, the MDG system interprets its statements, written in MDG-HDL code, in parallel, unless otherwise stated.

An explicit clock *S* is defined in order to suppress the inherent parallelism of the MDG system (see Figure 3.1). *S* is defined to be of concrete sort *wordc2*.

3.4.2 Embedded Software Implementation

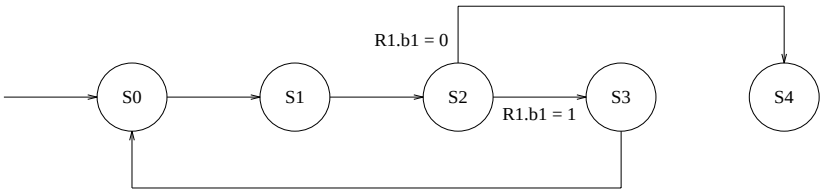
Assembly code:

```
00 START DECR R1
01       SETB R2.b2
02       SKBZ R1.b1
03       GOTO START
04       RESETB R2.b2
```

Notations:

```
DECR Ri        decrement and updates register Ri
SETB Ri.bj     sets bit j of register Ri
SKBZ Ri.bj     skip next instruction, if bit j of reg Ri is reset
GOTO  L        jump to the address indicated by the label L
RESETB Ri.bj   resets bit j of register Ri
```

ASM model:



MDG-HDL model:

```
initialize pc = 0
case pc = 0: next(pc) = 1
             R1 = decr(R1)
case pc = 1: next(pc) = 2
             R2 = setb(R2,b2)
case pc = 2: if testbitforzero(R1,b1) = 0
                then next(pc) = 4
                else next(pc) = 3
case pc = 3: next(pc) = 0
case pc = 4: R2 = resetb(R2,b2)
```

Figure 3.2: An example assembly code fragment and its MDG model

The implementation considered here is the assembly language code programmed (embedded) in a microcontroller. Figure 3.2 illustrates an example of assembly code program, its ASM model, and its description in MDG-HDL. The assembly code implementation is modeled as a set of instructions implementing the control behavior of a

routine. This is represented as an ASM, as shown by the state diagram in Figure 3.2. The ASM model has five states. Labels “00” to “04” refer to the five states of the ASM. Each state implements one instruction of the program. As is adopted in the behavior, this model uses the same abstract functions *decr* to decrement a register, and abstract functions *setb* and *resetb* to set and clear a particular bit of a register respectively, and the cross function *testbitforzero* to test if the value of a particular bit of a register is zero. The abstract sort *worda8* is used to model the registers *R1* and *R2*, and the abstract sort *worda3* is used to model the bits *b1* and *b2*. Here, the *pc*, of concrete sort *wordc3*, sequences the execution of each instruction (see Figure 3.2).

Chapter 4

Hierarchical Verification of Embedded Systems with MDGs

4.1 Introduction

The sequential equivalence checking, property checking and model checking procedures of the MDG system are used to carry out our verification experiments.

Sequential equivalence checking between the specification (or behavior) and the implementation is done to ensure that the implemented circuit or assembly code reflects its intended behavior. In the MDG software package, sequential equivalence is ensured when the outputs of the specification are the same as those of the implementation at every clock cycle [60]. This is achieved by feeding the same inputs to the two circuits, thereby forming a *parallel combination* of the circuits, which implements the *product machine* [23] [59].

Since a specification is an abstract behavior, it does not necessarily produce its outputs during the same clock cycles as its implementation does. A sequential ASM behavior would take lesser clock cycles than its implementation. Thus, we need a way to synchronize both. We achieve this by introducing a variable, *sync*, which keeps track of the execution of each instruction in the implementation (RTL hardware or the assembly code). This is passed as an input to the specification (ISA or the flowchart behavior), wherein it delays each of its output until its implementation is ready to produce the same output. This is done in a simple way in MDG-HDL by adding a new input column in each of the output table

descriptions in the specification, under the name *sync*. Also, it is mandatory in the MDG system that the specification and the implementation be required to use the same function symbols while carrying out equivalence checking [60].

Property verification is used for verifying that a design satisfies some specific requirements. Property verification in the MDG system can be achieved through invariant checking [60] and model checking [57].

4.2 Hierarchical Verification Approach

A bottom-up approach is adopted to verify the embedded system model. First the correctness of the microcontroller hardware (RT level) in implementing its instruction set architecture is verified. Having ensured the fact that each of the instructions in the instruction set is correctly executed in the hardware, it is then proceeded to verify the embedded assembly program against its behavioral specification. This paves a way for automatic verification of an embedded system hierarchically at different levels of abstraction.

4.2.1 Verification of the ISA

The instruction set architecture of a microcontroller is the specification of the effect that each instruction is intended to have on the visible state, which consists of the visible registers and memory. To verify a microcontroller implementation against its instruction set is to verify that the hardware execution of every instruction has the intended effect.

Let circuit *M* be the microcontroller architecture. *M* is compared with an ideal state machine *M'*, whose state is the visible state of *M* and where each transition corresponds to

the execution of an instruction as specified by the architecture. The control FSM of a conventional microcontroller has a distinguished ready state that is the starting point of instruction execution, which is signalled by the starting of the fetch cycle. This ready state is extracted from *M* and is used as the *sync* to achieve synchronization between *M* and *M'*: when *ready* = 1 the specified transition takes place in *M'*, otherwise *M'* remains in the same state. Using the sequential equivalence checking algorithm of the MDG package the respective contents of the program counter, program memory, instruction register and the general purpose registers between the RTL implementation (machine *M*) and the ISA (machine *M'*) are compared at each *ready* state [59].

4.2.2 Verification of the Embedded Software

4.2.2.1 Equivalence Checking

Verification of the embedded software is done by comparing the value of the flags and registers between the behavioral flowchart and the implemented assembly code program. The behavioral specification consumes less clock cycles than the assembly program. Hence, in the assembly implementation a variable *sync* is declared and is defined to be equal to the *pc*. It is an output from the implementation and is taken as an input for the specification. Below is shown the synchronized MDG model for the specification that is amenable to sequential equivalence checking:

Synchronized MDG-HDL model for the specification:

```
case s = 0: when(sync = 0) R1 = decr(R1)
            when(sync = 1) R2 = setb(R2,b2)
            next(s) = 1
```

```
case s = 1: if testbitforzero(R1,b1) = 0
                then next(s) = 2
                else next(s) = 0
case s = 2: when(sync = 4) R2 = resetb(R2,b2)
```

Thus, the outputs are checked for equivalence at each cycle of the implementation.

4.2.2.2 Invariant Checking

Property checking is used for verifying that a design satisfies some specific requirements. MDG tools allow checking of safety properties as invariants. Verification of invariants is directly based on the *reachability analysis* procedure of the MDG tools. Given a state machine *M* and an invariant *C*, it is checked if *C* holds in all reachable states of *M*.

Following are two properties of the embedded software (Section 3.4.1), expressed as invariants:

Property 1: **if** [¬ R1.b1] **then** [R2.b2=0]

Property 2: **if** [R1.b1] **then** [R2.b2=1]

Property 1 states that the flag *R2.b2* is reset, given the condition that bit *b1* of register *R1* is reset. Property 2 states that the flag *R2.b2* is set, given the condition that bit *b1* of register *R1* is set.

Each of the *if* and the *then* statements is first transformed into a *Directed Formula* (DF) [14]. A DF can be represented in terms of a circuit. Thus, a property is represented by its equivalent circuit form. The two circuits corresponding to a property are composed and run concurrently with the state machine on which the property is to be checked. The output of the circuit corresponding to the condition (*if*) statement is compared with the output of the

circuit corresponding to the *then* statements. The equivalence between the two is checked as an invariant.

4.2.2.3 Model Checking

Model checking is a technique to prove temporal properties on a design model under all possible and allowable conditions. In MDG model checking, the properties are described using a property specification language called L_{MDG}, which is a subset of a first-order branching time temporal logic that supports abstract data representations. Both safety and liveness properties can be expressed in L_{MDG}, however, only universal path quantification is possible.

Following are two properties of the embedded software (Section 3.4.1), expressed in L_{MDG}:

Property 1: `AG((R1.b1=0) -> (X(R2.b2=0)))`

Property 2: `AG((R1.b1=1) -> (X(R2.b2=1))),`

where *A* is the universal path quantifier, and *G*, *X* are state quantifiers. (*AG p*) implies "For all paths *p* is true in all states". (*X q*) implies "*q* is true in the succeeding state, from a given state".

Property 1 states that the flag *R2.b2* is reset in the succeeding state, given the condition that bit *b1* of register *R1* is reset in some state. Property 2 states that the flag *R2.b2* is set in the succeeding state, given that bit *b1* of register *R1* is set in some state.

The computation model is based on ASMs. To check a property *p* in L_{MDG} on an ASM *M*, first additional ASMs M_j for basic sub-formulas of *p* in which only the temporal operator X (next state) is allowed. These sub-formulas are called *Next_let_formulas*. In

general, it takes two steps to check a property expressed in L_{MDG}. The first step is to automatically build additional ASMs that represent the *Next_let_formulas* appearing in the property, and then connect these additional ASMs to the original ASM. The property parser developed in C, with Yacc&Lex is used for this step. The second step is to check a simpler property on the composite machine. The MDG Model checker developed in Prolog within the MDG package is used for this step. Further details on the MDG model checking procedure is found in [57].

Chapter 5

Case Study Application - PIC 16C71

The embedded system that is investigated in this work is the PIC16C71 microcontroller [46], commercialized by Microchip Technology Inc., and its mouse controller application software [45]. The hierarchical approach to modeling and verification is demonstrated on this target system and the experimental results are presented.

5.1 The Target Embedded System

5.1.1 Hardware RT-level Architecture

The PIC16C71X family of microcontrollers fits perfectly in applications ranging from security and remote sensors to appliance control and automotive. The EPROM technology makes customization of application programs extremely fast and convenient. Low cost, low power, high performance, ease of use and I/O flexibility make the PIC16C71X very versatile even in areas where no microcontroller use has been considered before (e.g. timer functions, serial communication, etc.).

The PIC16C71 is a low cost, high performance, 8-bit microcontroller, employing an advanced RISC-like architecture (Figure 5.1). There are 36 8-bit wide general purpose registers, 15 special function registers, a 13-bit wide program counter and a hardware stack. The hardware stack is 8-level deep and has 36 bytes of RAM. The separate instruction and data bus of the Harvard architecture allow a 14-bit wide instruction word with a separate 8-

bit wide data. An instruction cycle consists of eight Q cycles (Q1 to Q8). A fetch cycle begins with the program counter (PC) incrementing in Q1. The instruction is fetched from the program memory and latched into the instruction register in Q5. This instruction is then decoded and executed during the Q6, Q7, and Q8 cycles. Data memory is read during Q6 (operand read) and written during Q8 (destination write). All instructions execute in a single instruction cycle except for program branches.

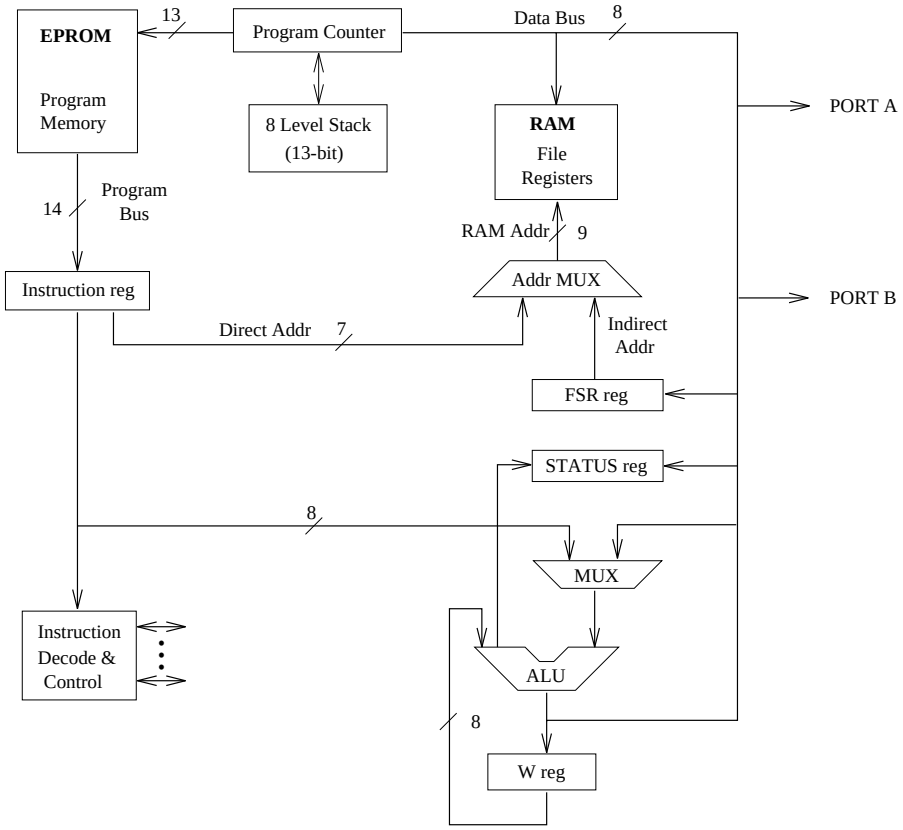


Figure 5.1: Microcontroller architecture (RT-level)

5.1.2 Instruction Set Architecture

The instruction set architecture (ISA) (Appendix A) consists of a total of 35 instructions (reduced instruction set). The instruction set is highly orthogonal, that makes it possible to

carry out any operation on any register using any addressing mode. The instruction set is categorized into four general formats of operations (Figure 5.2):

- byte-oriented (instructions acting on two registers)
- literal (instructions acting on the working register, *w*)
- bit-oriented (instructions acting on one bit of a register)
- control (instructions acting on the program counter, *pc*)

Each instruction is 14-bit wide and is divided into an OPCODE, which specifies the instruction type and one or more OPERANDS, which further specify the operation of the instruction. For byte-oriented instructions, *f* represents a file register designator and *d* represents a destination designator. The file register designator specifies which file register is to be used by the instruction. The destination designator specifies where the result of the operation is to be placed. If *d* is zero the result is placed in the *w* register and if it is one the result is placed in the *f* register specified in the instruction. For bit-oriented instructions, *b* represents a bit field designator which selects the number of the bit of the *f* register to be used by the instruction. For literal operations, *k* represents an 8-bit immediate value. For control operations *k* represents a 11-bit address.

5.1.3 Mouse Controller Software

The mouse is becoming increasingly popular as a standard pointing data entry device. Various kinds of mice can be found on the market, such as optical mice, opto-mechanical mice or trackball mice. Their basic mechanisms are very similar. The major electrical components of a mouse are: Microcontroller, Photo-transistors, Infrared emitting diodes, and Voltage conversion circuit. The mouse can be divided into several functional blocks

(Figure 5.3): Control, Button detection, Motion detection, Interface signal generation (typically, RS-232), and DC power supply. The intelligence of the mouse is provided by the microcontroller, therefore the features and performance of a mouse is greatly related to the microcontroller and the embedded program used to implement the function.

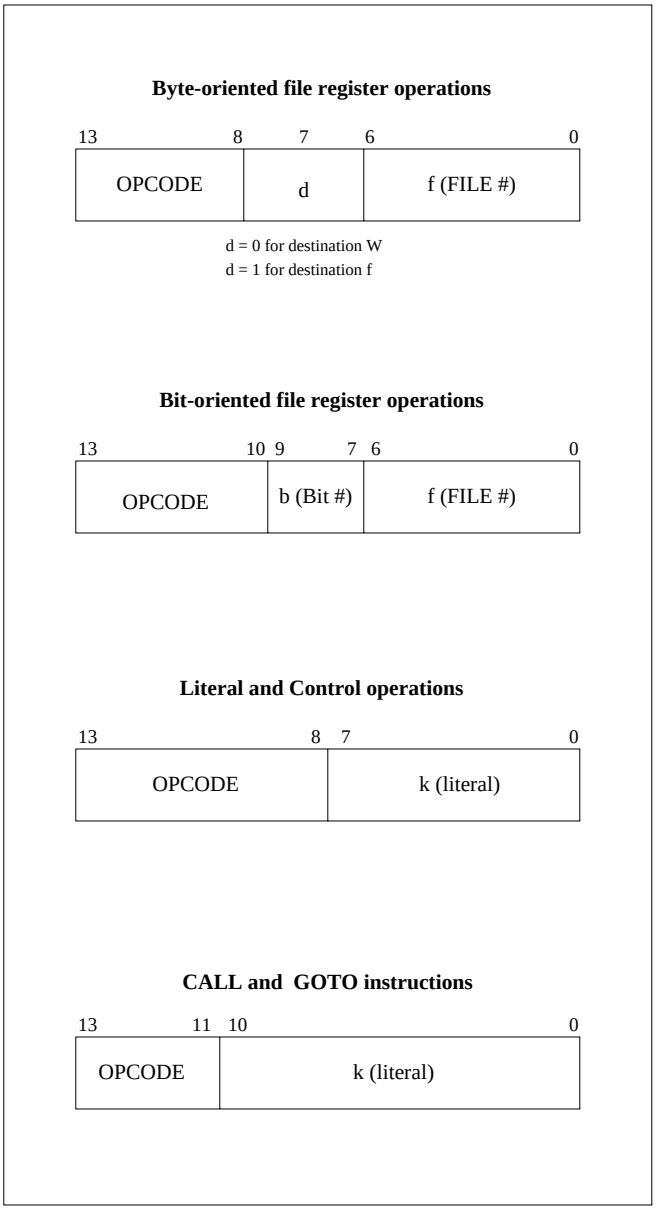


Figure 5.2: General format for instructions in ISA

In the following, the implementation of a serial mouse using the PIC16C71 [46, 45] is described. The major tasks performed by the embedded software are: Button scanning, X and Y motion scanning, and formatting and sending data to the host. To achieve the above mentioned goals the software is composed of three parts:

- *Main* program (Appendix B, Figure 1)
- Subroutine *Byte* (Appendix B, Figure 2)
- Subroutine *Bit* (Appendix B, Figure 3)

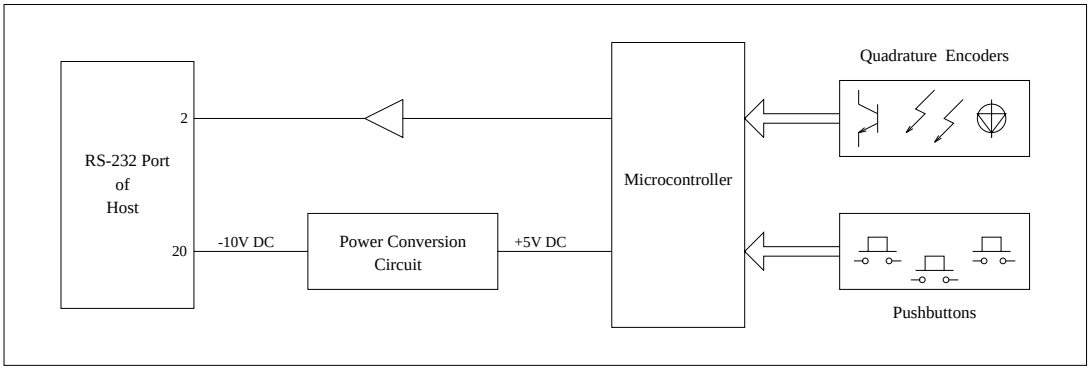


Figure 5.3: Functional blocks of a serial mouse

The *Main* program detects any changes in the button status and in the movement counts and sets a *Trigger flag*. The *Main* program calls two subroutines: *Byte* and *Bit*. The *Main* calls the *Byte* five times to send five bytes of data. The *Byte* calls the subroutine *Bit* periodically. The *Byte* converts the parallel data formatted in the *Bit* into a serial data on the "Received Data" (*RD*) pin and controls the status of *RD*. If *Trigger flag* is clear *RD* will always be high and no message will be sent even when *Byte* is called. The *Bit* counts the number of pulses from the outputs of the photo detectors and determines the direction of movement.

The routine *Bit* has two subroutines *Bitx* and *Bit*y. The subroutine *Bitx* tracks the right and left movement of the mouse and *Bit*y tracks the up and down movement. A right movement is detected when *XData* (*XD*) is zero during a positive edge of the *XClock* (*XC*) or when *XD* is one during a negative edge of *XC*. The *Bit1* section of *Bitx* detects the former condition for a right movement (*XD* being zero during a positive edge of *XC*). A *Right Flag* being set indicates a movement to the right, and the *XCount* gives the extent of the right movement. The section *Bit0* detects the latter condition for a right movement (*XD* being one during a negative edge of *XC*). Similarly, an up movement is detected when *YData* (*YD*) is zero during a positive edge of the *YClock* (*YC*) or when *YD* is one during a negative edge of *YC*. The *Bit0* and *Bit1* sections of *Bit*y detects the two conditions for an up movement respectively and accordingly set an *Up Flag*.

5.2 Hierarchical Modeling of the Target System

5.2.1 RTL Architecture

The RTL netlist implementation of the microcontroller (Figure 5.1) is described using MDG-HDL. A hierarchical description down to the MDG-HDL library of basic components (see Chapter 2) is adopted. The 8-bit general purpose registers are modeled as variables of abstract sort *worda8*. The register file is described in terms of uninterpreted access functions *read* and *write*, with the particular register to be accessed being supplied as an argument to the functions. The uninterpreted function symbol *fetch* is used to model the instruction fetch from the program memory. The ALU functions are expressed using uninterpreted function symbols *increg*, *decreg*, *iorreg*, *setbit*, *clearbit*, *testbit*, etc. The system bus is modeled using the basic component *driver*. The 13-bit program counter is

modeled as a variable of abstract sort *worda13*. The Q cycles of an instruction are modeled using a variable of concrete sort *wordc4*, with enumeration {0,...., 15}, which can accommodate the 8 cycles.

5.2.2 Instruction Set Architecture

The instructions in the instruction set of PIC 16C71 microcontroller (Appendix A) are modeled as predicates using MDGs. The operations in the instructions are modeled using uninterpreted functions, applied to the arguments of the instructions (predicates) as follows: byte oriented operations as *increg*, *decreg*, *iorreg*, *movreg*, *clearreg*, etc., bit oriented operations as *setbit*, *clearbit*, *testbit*, etc., literal operations as *addlitw*, *andlitw*, *iorlitw*, etc., and control operations as *call*, *goto*, *return*, etc. The instruction fetch is modeled using the uninterpreted function *fetch*. Decoding the operation, source and the destination operands from the instruction are modeled using the uninterpreted functions *decode_opcode*, *decode_src_op*, and *decode_destn_op* respectively.

The model is illustrated below using the following two instructions: the instruction that sets a particular bit (b) of a register (R) (BSF R b) [46], and the unconditional branch instruction (GOTO K) [46]:

Assembly Instruction: `BSF R b`

MDG-HDL Model:

Definitions: `var(R, worda8)`

`var(b, worda3)`

`function(`**`setbit`**`, inputs[worda8, worda3], output[worda8])`

Instruction: `transform(inputs([R, b]), function(`**`setbit`**`), output(R))`

Assembly Instruction: `GOTO K`

MDG-HDL Model:

Definitions: `var(K, worda11)`

`var(pc, worda13)`

`function(goto, input[worda11], output[worda13])`

Instruction: `transform(inputs([K]), function(goto), output(pc))`

5.2.3 Embedded Software Specification

The specification for the routine *Main* and the two subroutines *Byte* and *Bit* are derived from their respective algorithmic flowcharts (Appendix B, Figures 1, 2, 3). They are modeled as ASMs, given mainly by tabular representation of the transition and output relations.

The state diagrams of the specification of the *Main*, *Byte* and *Bit* routines (Appendix B, Figures 4, 5, 6) consists of 13, 11, and 17 states, respectively. The model uses the abstract sort *worda8* for representing the registers, *BSTAT*, *Counter*, *XCount*, *YCount*, *Data*, and *RB*. Concrete sort *wordc3* is used to address a particular bit in a register. The concrete sort *bool* is used to specify (the value of) a particular bit of a register, viz., *XClock*, *YClock*, *XData*, *YData*, *RD*, *Triggerflag*, *Rightflag* and *Upflag*. The uninterpreted abstract functions *bitset* [$worda8 \times worda3 \rightarrow worda8$] and *bitclear* [$worda8 \times worda3 \rightarrow worda8$] are used to set and reset a specific bit of a register respectively. A cross function *isregzero* [$worda8 \rightarrow bool$] is used to test whether or not the contents of a register is reset. Similarly the cross function *isbitzero* [$worda8 \times worda3 \rightarrow bool$] is used to test if a particular bit of a register

is reset. The concrete sort *wordc4* is used to represent the values of the clock *S* in routines *Main* and *Byte*, and *wordc5* is used to represent the values of *S* in the routine *Bit*.

5.2.4 Embedded Software Implementation

The specification of the routine *Main* and the two subroutines *Byte* and *Bit* are implemented using the assembly language of PIC 16C71. The assembly implementation is modeled as a set of instructions implementing the control behavior of the routines. They are modeled as ASMs, given mainly by tabular representation of the transition and output relations. The MDG ASM model consists of as many states as there are instructions, each for implementing one instruction at a time. Thus the enumeration of *pc* is equal to the number of instructions in the routine. Since the model is too large to include here, the model of the *Bit1* section of the *Bit* routine is shown in Fig. 5.4. Furthermore, the model uses the same constants, and abstract and cross functions as in the specification model, which is mandatory for equivalence checking using MDG tools.

5.3 Hierarchical Verification of the Target System

In this section, the bottomup approach for the hierarchical verification of the target embedded system model is described, using the MDG tools. This section also shows how the model is validated using the MDG tools. The experiments were conducted on a SUN SPARC ULTRA 1 with 256 MB of main memory. The results are summarized in tables, showing the performance statistics of the verifications, including CPU time, memory usage and number of MDG nodes generated. The CPU time is the time used for compiling the

circuit descriptions and for *reachability analysis*, including counterexample generation, if necessary.

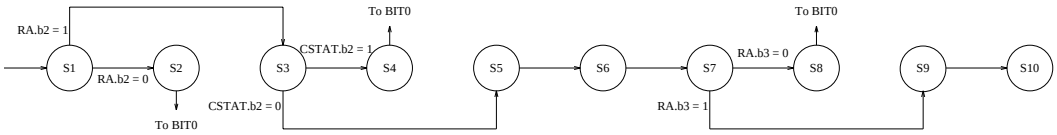
Assembly language program:

```
01: BTFSS RA.b2
02: GOTO BIT0
03: BTFSC CSTAT.b2
04: GOTO BITY
05: INCF XCOUNT
06: BCF FLAGB.b3
07: BTFSS RA.b3
08: GOTO BITY
09: BSF FLAGB.b3
10: GOTO BITY
```

Notations:

```
BCF Ri.bj  : clear bit j of register Ri
BSF Ri.bj  : set bit j of register Ri
```

ASM model:



MDG-HDL model:

```
initialize pc = 0
case pc = 1 :  if (RA.b2 = 0) then next(pc) = 2
               else next(pc) = 3
case pc = 2 :  goto(BIT0)
case pc = 3 :  if (CSTAT.b2 = 1) then next(pc) = 4
               else next(pc) = 5
case pc = 4 :  goto(BITY)
case pc = 5 :  inc(XCOUNT)
               next(pc) = 6
case pc = 6 :  reset(FLAGB.b3)
               next(pc) = 7
case pc = 7 :  if (RA.b3 = 0)then next(pc) = 8
               else next(pc) = 9
case pc = 8 :  goto(BITY)
case pc = 9 :  set(FLAGB.b3)
               next(pc) = 10
case pc = 10:  goto(BITY)
```

Figure 5.4: ASM and MDG model of *Bit1* of *Bitx* routine implementation

5.0.1 Verification of the ISA

The contents of the program counter, program memory, instruction register, working register, and the general purpose registers in the RTL architecture model are compared to the respective in the instruction set model at every cycle Q = 1, which is the ready state of the target microcontroller. The above mentioned comparison is demonstrated using the MDG equivalence checking procedure.

The following are the instructions that are used in the mouse controller program: INCF, BSF, BCF, BTFSS, DECFSZ, BTFSC, GOTO, CALL and RETLW, which span all the four categories of operation in the instruction set. The hardware of the target microcontroller is verified to be implementing the above mentioned instructions in its instruction set. The performance statistics of the verification are given in Table 5.1.

Verification	Result of Verification	CPU time (seconds)	Memory usage (MB)	No. of MDG Nodes
RTL against ISA	*successful*	14.27	9.15	21940

Table 5.1: Performance statistics of microcontroller hardware verification

5.0.2 Equivalence Verification of the Embedded Software

A hierarchical approach is followed for the verification of the mouse controller embedded software program. To start with, the subroutines *Bitx* and *Bity* of the routine *Bit* are verified. The verified subroutines are then abstracted away and replaced by their respective specifications, which have less states than their respective implementations, shown in Figure 5.5. The routines *Bit*, *Byte* and *Main* are verified hierarchically in this manner, using the MDG equivalence checking procedure. This approach effectively reduces the state space of the product machine and hence the verification CPU time.

Further simplification is possible if the state of the calling routine, after the return from the called routine, does not depend on the outputs of the called routine. In other words, if the state of the calling routine is not changed by the called routine, then each of the routines could be verified separately and independently of each other. In instances where this simplification could be applied, it tremendously reduces the state space, and the models become completely modular. Interested readers are referred to Appendix C for a formal proof of the approach.

Futhermore, the routine *Bit* is called eight times in the routine *Byte*, and the routine *Byte* is called five times in routine *Main*. Once a routine is verified, it is redundant to verify every time it is being called by another routine, since the verification implicitly checks the outputs of a routine for all combinations of its inputs in all reachable states of the routine.

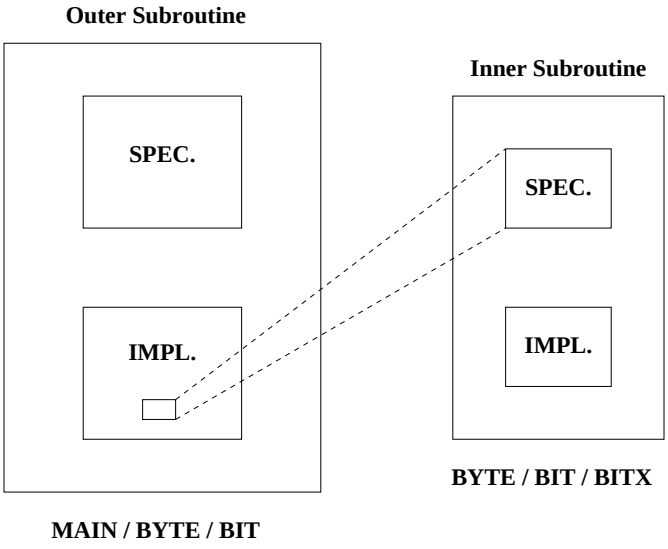


Figure 5.5: Hierarchical Verification Approach

Using the hierarchical approach, the equivalence between the behavioral specification and the embedded assembly software of the Main, Byte and the Bit routines are verified.

The equivalence checking of *Bit0* of the routine *Bit* showed an error in the assembly language code. This was indicated by a counterexample generated by the MDG tools, providing a trace leading to the software error. The error was found to be located in the instruction 6 (Figure 5.4), the instruction 6 is to be BTFSC in place of BTFSS. This result confirms with the one obtained in [54] using SMV. The erroneous instruction is corrected in no time and the equivalence checking is run successfully on the subroutine *Bit0*. The rest of the routines are verified successfully. Table 5.2 shows the performance statistics of the equivalence verification (behavioral specification against assembly code implementation) of each of the routines.

Verification of	Result of Verification	CPU time (seconds)	Counterex. gen. (seconds)	Memory usage (MB)	No. of MDG Nodes
BIT0 of BITX	*successful*	0.12	-	1.33	783
BIT1of BITX (org)	*failed*	0.18	0.43	1.30	918
BIT1of BITX (corr)	*successful*	0.11	-	2.05	783
BITX	*successful*	0.10	-	7.57	357
BIT0 of BITY	*successful*	0.21	-	1.33	783
BIT1 of BITY	*successful*	0.17	-	1.66	783
BITY	*successful*	0.07	-	7.57	357
BIT	*successful*	1.00	-	2.56	3624
BYTE	*successful*	21.8	-	1.04	26345
MAIN	*successful*	25.67	-	6.11	38167

Table 5.2: Performance statistics of embedded software Verification

5.3.3 Invariant Checking of the Embedded Software

Properties are derived to test the status of the flags, in accordance with the primary inputs, from the embedded software specification (Section 5.2.3). Below is shown two properties of the *RightFlag*, expressed as invariants:

Property 1: **if** [RA.b2] $\wedge$ [$\neg$ CSTAT.b2] **then** [$\neg$ Rightflag]

Property 2: **if** [RA.b2] ∧ [¬ CSTAT.b2] ∧ [RA.b2] **then**

[Rightflag]

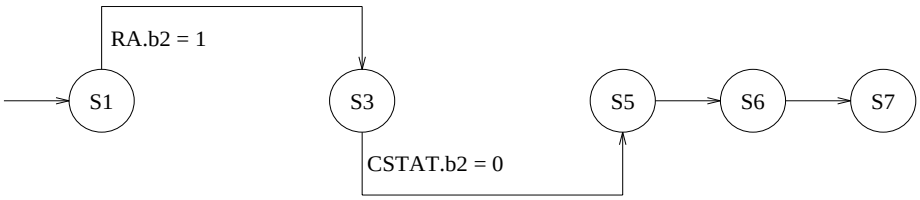


Figure 5.6: State machine corresponding to inputs RA.b2 = 1, CSTAT.b2 = 0

Property 1 checks whether the *Right Flag* is reset, when the inputs *RA.b2* = 1 and *CSTAT.b2* = 0. The environment state diagram for Property 1 is shown in Fig. 5.6. The state machine in Fig. 5.6 is an instance of the state machine in Fig. 5.4, given the environment that *RA.b2* = 1 and *CSTAT.b2* = 0. Property 2 checks whether the *Right Flag* is set, when inputs *RA.b2* = 1, *CSTAT.b2* = 0 and *XDATA* = 0. Property 2 is similar to the CTL property verified in [54].

The properties stated above of the *Right Flag* are checked on the flowchart specification as well as the embedded assembly language implementation of the routine *Bit*, using invariant checking procedure of the MDG tools. The Property 1 succeeded on the specification and the implementation state machines. Property 2 failed to succeed on the implementation machine. The MDG tool generated a counterexample during the invariant checking, which confirmed with the error located by the equivalence checking in Section 5.0.2. The instruction 6 was corrected and the property 2 was checked to hold good on the corrected implementation machine. Table 5.3 shows the performance statistics for the invariant checking.

Property	Verification on BIT	Result of Verification	CPU time (seconds)	Counterex. gen. (seconds)	Memory usage (MB)	No. of MDG Nodes
property 1	org. impl	*successful*	1.37	N/A	1.08	592
property 1	cor. impl	*successful*	1.34	N/A	1.06	567
property 1	spec	*successful*	1.31	N/A	1.06	567
property 2	org. impl	*failed*	1.44	0.10	1.10	711
property 2	cor. impl	*successful*	1.37	N/A	1.08	613
property 2	spec	*successful*	1.46	N/A	1.07	588

Table 5.3: Performance statistics of invariant checking on *Bit*

5.3.4 Model Checking of the Embedded Software

Model checking experiments were conducted to test the temporal properties of the embedded software, using the MDG model checking procedure. Furthermore, the model checking experiment given in [54] was re-verified using a recent version of the SMV tool, running on a faster machine [47]. This experiment was conducted to have a comparison with the MDG model checking experiments in terms of time and complexity. The experiments were performed on SUN SPARC ULTRA 1 with 256 MB of main memory.

5.3.4.1 Model Checking using MDG Tools

Temporal properties of the embedded software are derived from the embedded software specification (Section 5.2.3). Two of the properties that tests the status of the *Rightflag*, expressed in L_{MDG} are as follows:

Property 1: `AG((RA.b2=0 & CSTAT.b2=1) -> (X(Rightflag=0)))`

Property 2: `AG((RA.b2=0 & CSTAT.b2=1 & XDATA=1) -> (X(Rightflag=1)))`

Property 1 checks whether the *Right Flag* is reset in the succeeding state, after the inputs *RA.b2* = 1 and *CSTAT.b2* = 0 are assigned. Property 2 checks whether the *Right Flag* is set

in the succeeding state, after inputs *RA.b2* = 1, *CSTAT.b2* = 0 and *XDATA* = 0 are assigned. Property 2 is similar to the CTL property verified in [54].

The model checking experiments are performed on the specification and the implementation models of the subroutine *Bit1* of *Bitx*, to verify if the models confirm to the above mentioned properties. The model checking of Property 2 on the implementation model of the subroutine indicated a failure. The error was traced manually to be found in instruction 6, and was corrected. The experiment was repeated successfully on the corrected model of the subroutine, which confirmed with that shown in Section 5.0.2. Table 5.4 summarizes the performance statistics of the model checking experiments using MDG model checking.

Property	Verification on Bit1 of Bitx	Result of Verification	CPU time (seconds)	Memory usage (MB)	No. of MDG Nodes
property 1	org. impl	*successful*	0.160	1.48	1172
property 1	cor. impl	*successful*	0.170	1.41	1175
property 1	spec	*successful*	0.060	0.94	628
property 2	org. impl	*failed*	0.250	1.51	1323
property 2	cor. impl	*successful*	0.182	2.35	1250
property 2	spec	*successful*	0.130	0.98	613

Table 5.4: Performance statistics of MDG model checking

5.3.4.2 Comparison with Model Checking using Cadence SMV

Cadence SMV [43] has been found to be quite effective in automatically verifying properties of combinational logic and interacting finite state machines. It is a formal verification system based on symbolic model checking [11]. It uses a Verilog hardware description language to express the system model. The model is described in the boolean level. The specification is expressed as CTL formulas. SMV uses the OBDD symbolic

model checking algorithm to verify each of the CTL formulas on the Verilog model. SMV also has a counterexample generation feature.

Temporal properties of the embedded software are derived from the embedded software specification (Section 5.2.3). Two of the properties that tests the status of the *Rightflag*, expressed in CTL are as follows:

Property 1: `AG((RA.b2=0 & CSTAT.b2=1) -> F (Rightflag=0))`

Property 2: `AG((RA.b2=0 & CSTAT.b2=1 & XDATA=1) -> F (Rightflag=1)),`

where *A* is the universal path quantifier, and *G*, *F* are state quantifiers. (*AG p*) implies "For all paths *p* is true in all states". (*F q*) implies "*q* is true for one state in the future, from a given state". Property 1 checks whether the *Right Flag* is reset in one state in the future, after the inputs *RA.b2* = 1 and *CSTAT.b2* = 0 are assigned. Property 2 checks whether the *Right Flag* is set in one state in the future, after inputs *RA.b2* = 1, *CSTAT.b2* = 0 and *XDATA* = 0 are assigned. Property 2 is similar to the CTL property verified in [54].

The model checking experiments are performed on the implementation model of the subroutine *Bit1* of *Bitx*, to verify if the model confirms to the above mentioned properties. The model checking of Property 2 on the implementation model of the subroutine indicated a failure. The counterexample trace indicated an error found in instruction 6. The error was subsequently corrected and the experiment was repeated successfully on the corrected model of the subroutine, which confirmed with that shown in Section 5.0.2. Table 5.5 summarizes the performance statistics of the model checking experiments using SMV model checking.

BDD-based symbolic model checking requires the design to be described at the boolean logic level, the state explosion caused by large datapath is often the bottleneck in applying

symbolic model checking techniques. The MDG model checking raises the level of abstraction, in which the model is described using ASMs, which are encoded using MDGs. The verification of ASMs is based on state enumeration, the complexity of which is independent of the width of the datapath.

Comparing the results obtained in Table 5.5 with that in Table 5.4, a remarkable reduction in the size of the graphs and the CPU time is obtained with using MDGs. Furthermore, Thiry and Claesen [54] report the verification of Property 2 on the *Bit* routine, run on a 486DX33 machine with 16 MB RAM. The verification time reported was 23 seconds. Table 5.2 shows the equivalence verification of the *Bit* routine consumed only 1 second of CPU time. This demonstrates the ease and effectiveness in using MDG tools.

Property	**Verification on Bit1 of Bitx**	**Result of Verification**	**CPU time (seconds)**	**No. of Nodes**
property 1	org. model	*successful*	2.67	65721
property 1	cor. model	*successful*	2.66	65692
property 2	org. model	*failed*	4.79	77662
property 2	cor. model	*successful*	2.84	62821

Table 5.5: Performance statistics of SMV model checking

Chapter 6

Conclusions

While formal verification is an improvement over testing, it is only as good as the specification used and the soundness of the proof. Also, the description of the target systems, be it hardware or software, are only abstractions of the actual physical systems, and are thus subject to error or over-simplification. While correctness cannot be guaranteed, formal verification will nonetheless result in more reliable systems, reduced maintenance, and better quality control. Formal verification allows for objective, systematic review, because the formal descriptions, and the proof trace provide a permanent record of why the designer thinks the system is correct.

Embedded systems are gaining widespread applications by the fact that they allow for more flexibility and reconfigurability (the degree of which depends on the degree of the partition) and reduced design cycle time. This work attempts to emphasize the imperative need for verifying the reliability of such a system. It presents a hierarchical approach to model and verify an embedded system at different levels of design abstraction, using the MDG tools. The approach is demonstrated on a commercial microcontroller used in a mouse controller application. Models are established for the RT level hardware and the Instruction Set Architecture of the microcontroller, and the behavioral specification and its implementation as the assembly code of the embedded mouse controller application software, are presented using MDGs. Experiments are conducted using the equivalence checking, property checking and the model checking features provided by the MDG tools,

to verify the correctness of the hardware in implementing its ISA, and the correctness of the embedded software in implementing its specification. The verification experiments conducted on our models using the MDG tools concluded in few seconds of CPU time.

There are two essential differences between the approach proposed in this thesis and that presented in [54]. In [54] the authors present a single model that simulates the execution of the instructions in the embedded software on the RTL hardware, and verify a property on this single model. The proposed hierarchical approach splits the system into four models, each abstracted at different levels of the design hierarchy. This, in turn, splits the verification task into two separate verification experiments, thus rendering the verification approach less redundant, more modular, and easier debugging in case of errors. Moreover, the single model in [54] is represented at the boolean level, using ROBDDs, while the models presented in this thesis are elevated to higher levels of abstraction and represented using MDGs. From the experimental point of view, [54] demonstrates the approach on a single routine of an embedded software program. This thesis demonstrates the approach on a complete embedded system. The relatively small CPU time and memory consumption achieved in all the experiments demonstrate the efficiency of the use of abstract data types and uninterpreted functions, as provided by MDGs, to handle the inherent complexities of verifying a complete embedded system, in an automated environment.

At the same time, abstraction might lead to possible non-termination of state-exploration. Generalization of the initial states, in certain cases, counteracts this problem. A limitation from the application software point of view is that, an increase in the number of conditional branches increases the chances of state explosion. Since the *pc* of a software program has to be of concrete sort and can never be made abstract, this thesis shows two

ways to reduce the state space per verification: one is to verify each of the routines in a program separately, under certain conditions as stated in Section 5.0.2, and the other more generic way is to replace the implementation of the verified routines with their respective specifications in the calling routine. This shows that the MDG tool is more adapted to hardware verification, and need some improvement to handle the enormity of the states that could be found in a software program.

Representing our models in standard hardware description languages, namely VHDL, which has been recently interfaced to the MDG tools, would further enhance the applicability of the MDG tools for the verification of embedded systems and integrating it into the hierarchical design process. This will also eliminate possible human errors during hand translation of the design model into MDG-HDL model.

Bibliography

[1] S. Balakrishnan, S. Tahar. Modeling and Formal Verification of a Commercial Microcontroller for Embedded System Applications. Proc. IEEE 10th International Conference on Microelectronics (ICM'98), Monastir, Tunisia, December 1998, pp. 107-110.

[2] S. Balakrishnan, S. Tahar. A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs. Proc. IEEE 9th Great Lakes Symposium on VLSI (GLS-VLSI'99), Ann Arbor, Michigan, USA, March 1999, IEEE Computer Society Press, pp. 284-287.

[3] H. Barringer. Symbolic Verification of Hardware Systems. Proc. of IFIP Conference on Hardware Description Languages and their Applications (CHDL'95), Shiba, Japan, August 1995.

[4] H. E. Berg, W. E. Boebert, W. R. Franta, T. G. Moher. Formal Methods of Program Verification and Specification, Prentice-Hall Inc., 1982.

[5] R. S. Boyer, J. S. Moore. A Computational Logic Handbook. Academic Press, Boston, 1988.

[6] R. K. Brayton *et.al*. VIS: A System for Verification and Synthesis. Technical Report UCB/ERL M95, Electronics Research Laboratory, University of Berkeley, December 1995.

[7] B. C. Brock, W. A. Hunt, Jr.. Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP. Proc. IEEE International Conference on Computer Design (ICCD'97), Austin, October 1997, pp. 31-36.

[8] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986, pp. 677-691.

[9] R. E. Bryant, Y. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. Proc. 32nd ACM/IEEE Design Automation Conference (DAC'95), San Francisco, June 1995.

[10] K. Buchenrieder, A. Sedlmeier, C. Vieth. HW/SW Co-Design with PRAM's using CODES. Proc. Computer Hardware Description Languages and their Applications (CHDL'93), Elsevier Science Publishers B. V., 1993, pp. 65-78.

[11] J. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill. Sequential Circuit Verification using Symbolic Model Checking. Proc. 27th ACM/IEEE Design Automation Conference, IEEE Computer Society Press, Los Alamitos, June 1990, pp. 46-51.

[12] J. Burch, D. Dill. Automatic Verification of Pipelined Microprocessor Control, Computer Aided Verification, LNCS 818, Springer Verlag, 1994, pp. 68-80.

[13] R. Butler, G. Finelli. The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. Software Engineering Notes, Vol. 16, No. 5, December 1991, pp. 66-76

[14] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. Formal Hardware Verification. Methods and Systems in Comparison, LNCS 1287, State-of-the-Art Survey, Springer Verlag, 1997, pp. 79-113.

[15] E. M. Clarke *et.al*. Spectral Transformations for Large Boolean Functions with Application to Technology Mapping. 30th ACM/IEEE Design Automation Conference, Dallas, June 1993.

[16] E. Clarke, M. Fujita, X. Zhao. Hybrid Decision Diagrams. Proc. IEEE International Conference on Computer-Aided Design (ICCD'95), San Jose, California, Dallas, June 1995.

[17] B. Cohen, W. T. Harwood, M. I. Jackson. The Specification of Complex Systems. Addison-Wesley Publishing Company, 1986.

[18] A. Cohn, A Proof of Correctness of the Viper Microprocessor: The First Level, VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, Boston, 1988, pp. 27-71.

[19] A. Cohn, Correctness Properties of the Viper Block Models: The Second Level. Currents Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 1-91.

[20] A. Cohn, The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol. 5, May 1989, pp. 127-139.

[21] J. Cook. Verification of the C/30 Microcode using the State Delta Verification System (SDVS). Proc. 13th National Bureau of Standards, National Computer Sceurity Centre, October 1990, pp. 20-31.

[22] F. Corella, M. Langevin, E. Cerny, Z. Zhou, X. Song. State Enumeration with Abstract Descriptions of State Machines. Proc. of IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'95), October 1995, Frankfurt, Germany.

[23] F. Corella, Z. Zhou, X. Song, M. Langevin, E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. Formal Methods in System Design, Vol. 10, No. 1, 1997, pp. 7-46.

[24] D. Cyrluk, P. Narendran. Ground Temporal Logic: A Logic for Hardware Verification. Proc. Workshop on Computer-Aided Verification, 1994.

[25] R. Drechsler, B. Becker, S. Ruppertz. K*BMDs: A new Data Structure for Verification. Proc. IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'95), Frankfurt, October 1995.

[26] E. A. Emerson. Temporal and Modal Logic. Handbook of Theoretical Computer Science, Elsevier Science Publishers B. V., 1990, Chapter 16.

[27] R. J. Foulger. Programming Embedded Microprocessors: A High-level Language Solution. NCC Publications, Manchester, England, 1982.

[28] D. Gajski, et al. A System-design Methodology: Executable Specification Refinement. Proc. EDAC-94, Paris, 1994.

[29] D. Galter. Symbolic Verification of Instruction-Set Processors. Master of Mathematics Thesis, Dept. of Computer Science, University of Waterloo, 1994.

[30] A. Ghosh, S. Devadas, A. R. Newton. Sequential Logic Testing and Verification. Kluwer Academic Publishers, 1992.

[31] M. Gordon. LCF_LSM, Technical Report No. 41, Computer Laboratory, Cambridge University, 1983.

[32] M. J. C. Gordon. Proving a Computer Correct using the LCF_LSM Hardware Verification System, Technical Report No. 42, Computer Laboratory, Cambridge University, 1983.

[33] M. J. C. Gordon, T. F. Melham. Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge, U. K., 1993.

[34] A. Gupta. Formal Hardware Verification Methods: A Survey. Journal of Formal Methods in System Design, Kluwer Academic Publishers, Vol. 1, No. 2/3, 1992, pp. 151-238.

[35] W. A. Hunt. FM8501: A Verified Microprocessor. PhD Thesis, University of Texas, Austin, 1985.

[36] C. B. Jones. Systematic Software Development Using VDM. Prentice-Hall International, 1986.

[37] R. B. Jones, D. L. Dill. Efficient Validity Checking for Processor Verification. Proc. IEEE International Conference on Computer-Aided Design (ICCAD'95), San Jose, November 1995.

[38] J. J. Joyce. Multi-level Verification of Microprocessor-based Systems. Technical Report 195, Computer Laboratory, University of Cambridge, May 1990.

[39] M. Kaufmann, J. S. Moore. ACL2: An Industrial Strength of Nqthm. Proc. 11th Annual Conference on Computer Assurance (COMPASS'96), IEEE Computer Society Press, June 1996, pp. 23-34.

[40] A.Kundig, R. E. Buhrer, J. Dahler. Embedded Systems - New Approaches to their Formal Description and Design - An Advanced Course. Lecture Notes in Computer Science, Springer Verlag, Switzerland, No. 284, 1986.

[41] M. Langevin, E. Cerny. An Extended OBDD Representation for Extended FSMs. Proc. EDAC-ETC-EUROASIC, 1994.

[42] B. Littlewood, L. Strigini. Validation of Ultra-High Dependability for Software-based Systems, Communications of the ACM, Vol. 36, No. 11, November 1993, pp. 69-80.

[43] K. McMillan. Getting Started with SMV: User's Manual. Cadence Berkley Laboratories, USA, 1998.

[44] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[45] Microchip Technology Inc. "Embedded Control Handbook", 1993, pp. 2.121-2.133.

[46] Microchip Technology Inc., "PIC16C71", 1994, pp. 2.328-2.372.

[47] A. A. Mir, S. Balakrishnan, S. Tahar. Modeling and Verification of Embedded Systems using Cadence SMV. IEEE Canadian Conference on Electronics and Computer Engineering (CCECE'2000), Halifax, Novoscotia, Canada, May 2000.

[48] P. Narendran, J. Stillman. Formal Verification of the Sobel Image Processing Chip. Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989, pp. 92-127.

[49] S. Owre, J. Rushby, N. Shankar. PVS: A Prototype Verification System. Lecture Notes in Artificial Intelligence, Springer-Verlag, 1992, Vol. 607, pp. 748-752.

[50] J. Rushby. Quality Measures and Assurance for AI Software, Report No. SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, September 1988.

[51] M. Srivas, M. Bickford. Formal Verification of a Pipelined Microprocessor. IEEE Software, Vol. 7, No. 5, September 1990, pp. 52-64.

[52] M. K. Srivas, S. P. Miller. Applying Formal Verification to a Commercial Microprocessor. Proc. IFIP Conference on Hardware Description Languages and their Applications (CHDL'95). Chiba, Japan, August 1995.

[53] S. Tahar, R. Kumar. Implementing a Methodology for Formally Verifying RISC Processors in HOL. Higher Order Logic Theorem Proving and its Applications, LNCS 780, Springer Verlag, 1994, pp. 281-294.

[54] O. Thiry, L. Claesen. A Formal Verification Technique for Embedded Software. Proc. IEEE International Conference on Computer Design (ICCD'96), Austin, Texas, U. S. A, October 1996, pp. 352-357.

[55] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, A. Sangiovanni-Vindentelli. Implicit State Enumeration of Finite State Machines using BDDs. International Conference on Computer-Aided Design (ICCAD'90), San Francisco, California, U. S. A., November 1990.

[56] P. J. Windley. Formal Modeling and Verification of Microprocessors. IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp. 54-72.

[57] Y. Xu. MDG Model Checker User's Manual, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, September 1999.

[58] Y. Xu. Model Checking for a First-order Temporal Logic Using Multiway Decision Graphs. PhD Thesis, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, 1999.

[59] Z. Zhou. Multiway Decision Graphs and their Applications in Automatic Verification of RTL Designs. PhD. Thesis, Dept. of Information and Operational Research, Universite de Montreal, Montreal, Canada, 1997.

[60] Z. Zhou and N. Boulerice. MDG Tools (V1.0) User's Manual, Dept. of Information and Operational Research, University of Montreal, Montreal, Canada, 1996.

Appendix A

PIC 16C71 Instruction Set

Instruction		Description	Cycles	14-bit Opcode
Mnemonic	**Operands**			
Byte-oriented File Register Operations				
ADDWF	**f, d**	Add W with f	1	00 0111 dfff ffff
ANDWF	**f, d**	AND W with f	1	00 0101 dfff ffff
CLRF	**f**	Clear f	1	00 0001 1fff ffff
CLRW	**-**	Clear W	1	00 0001 0xxx xxxx
COMF	**f, d**	Complement f	1	00 1001 dfff ffff
DECF	**f, d**	Decrement f	1	00 0011 dfff ffff
DECFSZ	**f, d**	Decrement f, Skip if 0	1(2)	00 1011 dfff ffff
INCF	**f, d**	Increment f	1	00 1010 dfff ffff
INCFSZ	**f, d**	Increment f, Skip if 0	1(2)	00 1111 dfff ffff
IORWF	**f, d**	Inclusive OR W with f	1	00 0100 dfff ffff
MOVF	**f, d**	Move f	1	00 1000 dfff ffff
MOVWF	**f**	Move W to f	1	00 0000 1fff ffff
NOP	**-**	No Operation	1	00 0000 0xx0 0000
RLF	**f, d**	Rotate Left f through Carry	1	00 1101 dfff ffff
RRF	**f, d**	Rotate Right f through Carry	1	00 1100 dfff ffff
SUBWF	**f, d**	Subtract W from f	1	00 0010 dfff ffff
SWAPF	**f, d**	Swap nibbles in f	1	00 1110 dfff ffff
XORWF	**f, d**	Exclusive OR W with f	1	00 0110 dfff ffff
Bit-oriented File Register Operations				
BCF	**f, b**	Bit Clear f	1	01 00bb bfff ffff
BSF	**f, b**	Bit Set f	1	01 01bb bfff ffff
BTFSC	**f, b**	Bit Test f, Skip if Clear	1(2)	01 10bb bfff ffff
BTFSS	**f, b**	Bit Test f, Skip if Set	1(2)	01 11bb bfff ffff
Literal and Control Operations				
ADDLW	**k**	Add literal and W	1	11 111x kkkk kkkk
ANDLW	**k**	AND literal with W	1	11 1001 kkkk kkkk
CALL	**k**	Call Subroutine	2	10 0kkk kkkk kkkk
CLRWDT	**-**	Clear Watchdog Timer	1	00 0000 0110 0100
GOTO	**k**	Go to address	2	10 1kkk kkkk kkkk
IORLW	**k**	Inclusive OR literal with W	1	11 1000 kkkk kkkk
MOVLW	**k**	Move literal to W	1	11 00xx kkkk kkkk
RETFIE	**k**	Return from interrupt	2	00 0000 0110 1001
RETLW	**k**	Return with literal in W	2	11 01xx kkkk kkkk
RETURN	**-**	Return from Subroutine	2	00 0000 0000 1000
SLEEP	**-**	Go into standby mode	1	00 0000 0110 0011
SUBLW	**k**	Subtract W from literal	1	11 110x kkkk kkkk
XORLW	**k**	Exclusive OR literal with W	1	11 1010 kkkk kkkk

Appendix B

Mouse Controller Embedded Software Application

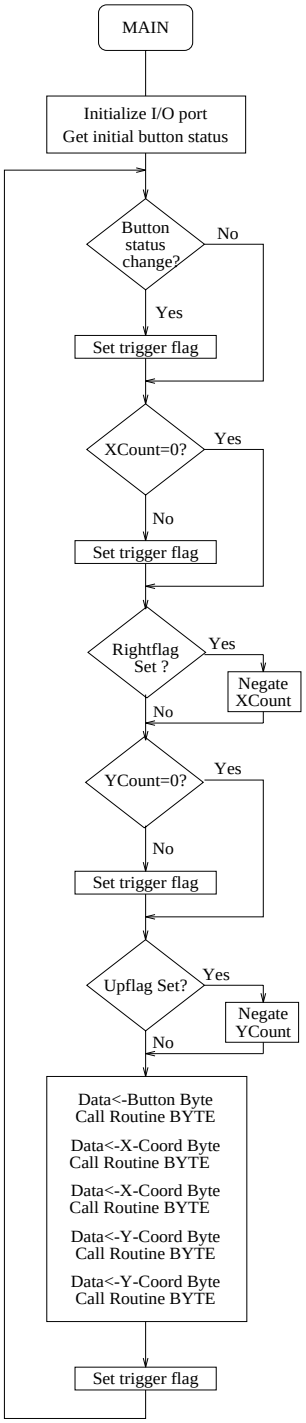


Figure 1: Flowchart specification of *Main* routine

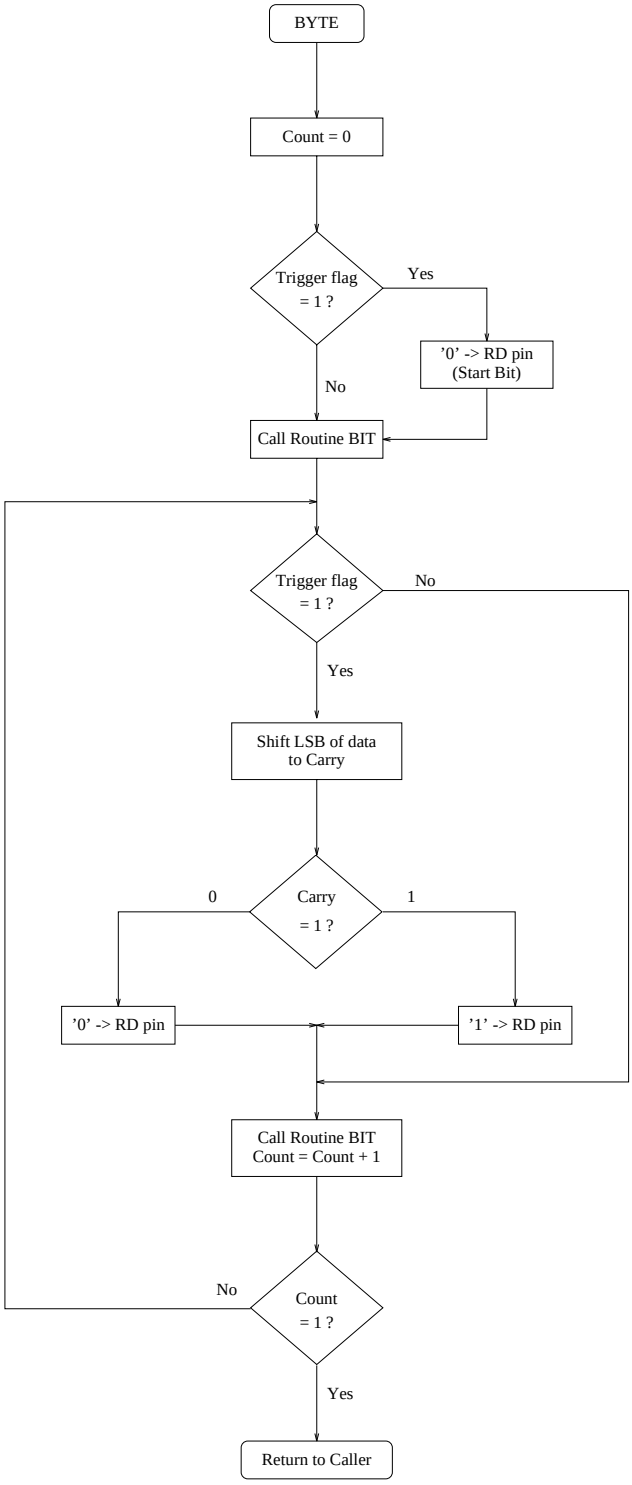


Figure 2: Flowchart specification of *Byte* routine

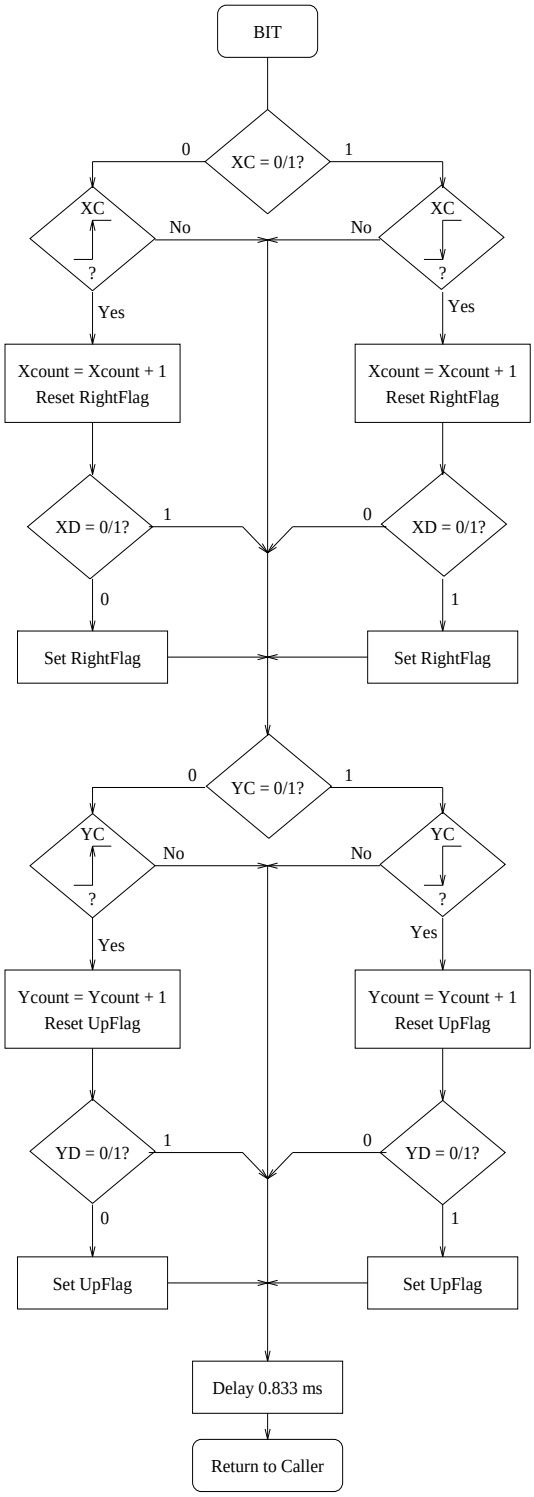


Figure 3: Flowchart specification of *Bit* routine

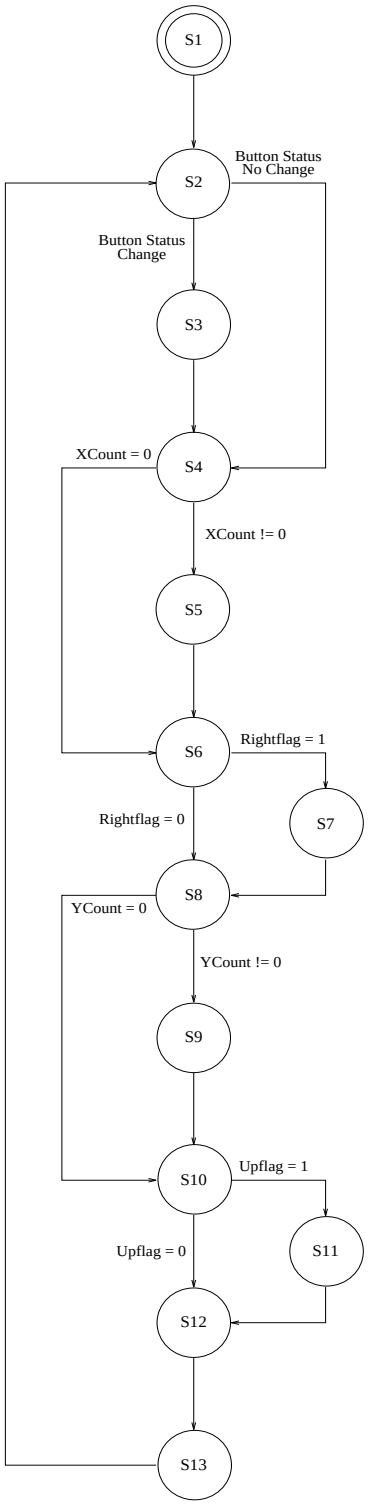


Figure 4: State diagram of specification of *Main* routine

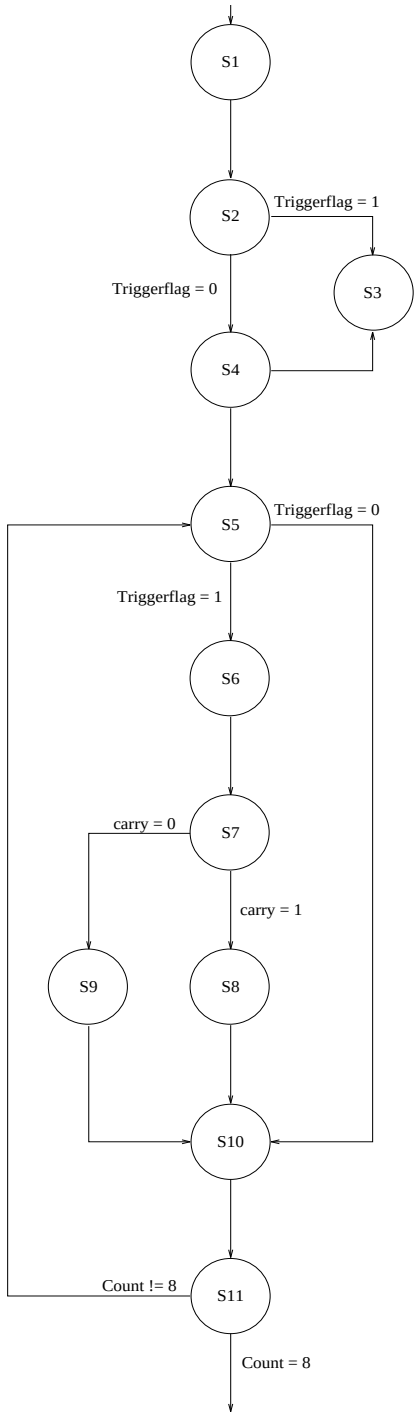


Figure 5: State diagram of specification of *Byte* routine

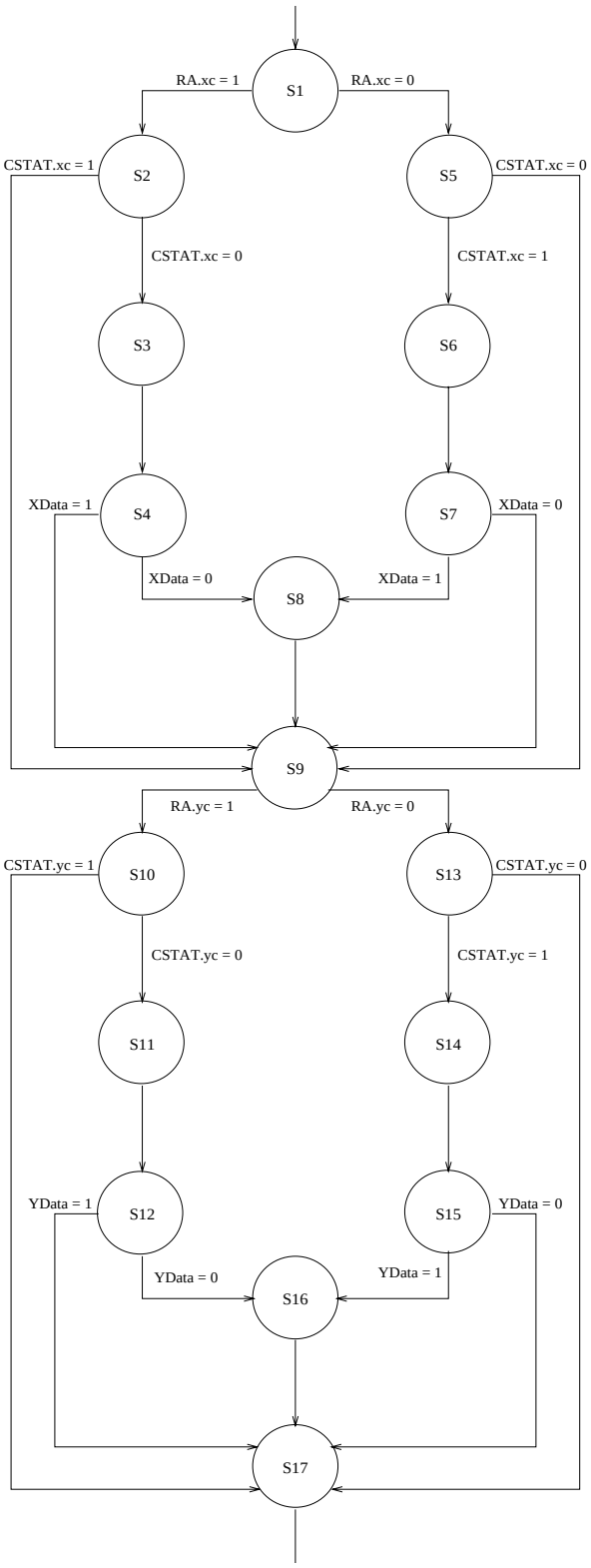


Figure 6: State diagram of specification of *Bit* routine

Appendix C

Proof of Hierarchical Approach

This section outlines a formal proof of the hierarchical approach to modeling and verification of embedded software. The approach is illustrated on a simple typical embedded software (Appendix C, Figure 1).

Let:

I be the set of all external inputs

O be the set of all outputs

S_1 be the set of all states of the calling routine

S_{10} be the state of the calling routine when calling the subroutine

S_{11} be the state of the calling routine when returning from the subroutine

i_1 be the set of all inputs to the calling routine

o_1 be the set of all outputs of the calling routine

i_2 be the set of all inputs to the subroutine

o_2 be the set of all outputs of the subroutine

Given that:

i_2 is $(S_{10} \cup i_1)$

o_2 is $(S_{10} \cup o_1)$

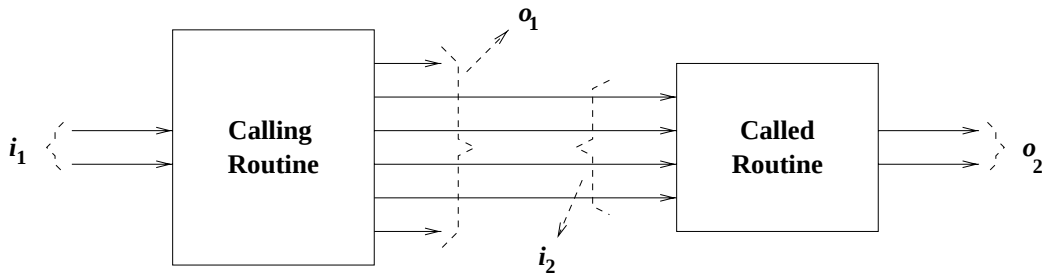


Figure 1: Generic black box representation of Embedded Software routines

To Prove:

$\forall I \; \forall S_1$ O is correct and S_{11} is correct

For all external inputs to the calling routine and the subroutine, and for all states of the calling routine, it is guaranteed that the new state of the calling routine and the external outputs are correct

Proof:

The routines are verified starting from the inner most subroutine, and successively replacing the behavior of the verified subroutines in the calling routine.

Verf.#1: $\forall i_2$ o_2 is verified

Verf.#2: $\forall i_1$ S_{10} is verified

Verf.#3: $\forall i_1$ S_{11} is verified

Deduction:

From Verf.#1 and Verf. #2,

$\forall S_{10}$ o_2 is verified, for the values that affect o_2

$\forall S_{10}$ that does not affect o_2, is don't care

From Verf.#3, and

Since, i_2 is $(S_{10} \cup i_1)$

o_2 is $(S_{10} \cup o_1)$

Thus, $\forall I$ $\forall S_1$ O and S_{11} are proved to be correct

When the state of the calling (outer) routine is unaffected by the called (inner) routine, then the verification and the proof process is further simplified as shown below:

Let:

i_1 be the set of all inputs of the routine *#1*

i_2 be the set of all inputs of the routine *#2*

i_3 be the set of all inputs of the routine *#3*

o_1 be the set of all inputs of the routine *#1*

o_2 be the set of all inputs of the routine *#2*

o_3 be the set of all inputs of the routine *#3*

Given that:

i_2 is a subset of o_1

i_3 is a subset of $(o_1 \cup o_2)$

To Prove:

$\forall$ i_1 o_3 is correct

Proof:

The routines are verified separately and independently of the other.

Verf.#1: $\forall$ i_3, o_3 is verified

Verf.#2: $\forall$ i_2, o_2 is verified

Verf.#3: $\forall$ i_1, o_1 is verified

Deduction:

From Verf.#1, Verf.#2, Verf.#3, and

Since, i_2 is a subset of o_1

i_3 is a subset of $(o_1 \cup o_2)$

Thus, $\forall$ i_1, o_3 is proved to be correct