Modeling and Verification of Embedded Systems using Cadence SMV

Ali Abbas Mir, Subhashini Balakrishnan and Sofiène Tahar

Technical Report

Electrical & Computer Engineering Department, Concordia University
1455 de Maisonneuve Blvd. W., Montreal, Quebec, H3H 1M8 Canada
Email: {aa_mir, subhar, tahar}@ece.concordia.ca

September 1999

Abstract. *Embedded systems are becoming increasingly popular due to their widespread applications. For safety-critical applications an approach is needed to validate the complexity of VLSI designs at a higher abstraction level. With formal verification we verify that every possible behavior of the target system satisfies the specification. SMV is a formal verification system for hardware designs, based on a technique called "symbolic model checking". It uses an OBDD (Ordered Binary Decision Diagram) algorithm to check whether CTL property specifications are met. In this report we investigate the modeling and verification of an embedded system using Cadence SMV. We constructed a Verilog model of the system by integrating the microcontroller RT level and the embedded software assembly code level. We then validate our models and verification by conducting model checking which analyzes essential aspect of the target embedded system.*

1 Introduction

Formal methods long have been touted as a means to produce provably correct implementations. It is only recently, however, with rather more modest claims, that one formal method: *model checking*, has been embraced by industry. This technology is blossomed from scattered pilot projects at a very commercial sites, into implementations in at least five commercially offered Design Automation tools. The use of formal methods to verify hardware design is termed formal hardware verification [5]. Formal hardware verification has recently attracted considerable interest. The need for correct designs in safety-critical applications, coupled with the major cost associated with products delivered late, are two of the main factors behind this. In addition, as the complexity of the designs increase, an even smaller percentage of the possible behaviors of the designs will be simulated. Hence the confidence in the designs obtained by simulation is rapidly diminishing. However, verification has the promise of reducing the simulation time, as well as increasing the level of confidence in the design. One of the formal design verification techniques is the process of validating a design by proving properties on it. The goal of this paper is to accomplish formal hardware verification of a microcontroller using Cadence version of SMV (Symbolic Model Verifier) [8] as a model checking tool.

In this report, we present a methodology and application of the formal verification of embedded software using the SMV tool. We model the instruction set of a commercial microcontroller [9] in the Cadence SMV input language, Verilog. We work out a method for the symbolic verification of the embedded software, in terms of assembly code, running on the above mentioned processor. We use the application of a mouse controller software with Microsoft compatible RS232 interface as an example to demonstrate our approach. Throughout our experiment, we uncovered inconsistencies between the specification and the implementation in the assembly code [9].

The rest of the paper is organized as follows. In Section 2, we discuss the related work done on embedded systems. In Section 3, we briefly introduce SMV (Symbolic Model Verifier). We describe the microcontroller architecture and its mouse controller software application in Section 4. In Section 5 we present the Verilog modeling of the RTL implementation. In Section 6, we discuss our formal verification approach along with experimental results. Section 7 concludes the paper.

2 Related Work

Recently, Embedded systems are finding widespread application including communication systems, factory automation, graphics and imaging systems, medical equipment and even household appliances. Technological advances in the areas of design and fabrication have made hardware systems much larger today than before. As faster, physically smaller and higher functionality circuits are designed, in large part due to progress made in VLSI, their complexity continues to grow.

In [13] a method for the verification of embedded software correctness was presented. A formal model for a commercial microcontroller, PIC16C71 [10] from Microchip Inc., was established. This was done by modeling the instruction set and processor architecture. Embedded software takes the form of assembly program code to be run on the processor. Specifications are given as CTL temporal logic formulae. The method had been implemented in the SMV (Symbolic Model Verifier) [7] model checker and was illustrated by a practical embedded system application: a mouse controller [9].

In [1], a hierarchical approach to modeling and formal verification of a complete embedded system at higher levels of abstraction, using Multiway Decision Graphs (MDGs) [3], is proposed. The approach is demonstrated on the embedded software for the same mouse controller application as in [13]. In difference to [13] however, the system is modeled and verified 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 first established by formally verifying the equivalence between the RTL hardware and the ISA, using the MDG sequential equivalence checking tool. Subsequently, the particular application embedded in the system is verified by checking the equivalence between the assembly code and its intended behavior, specified as a flowchart. Furthermore, safety properties and liveness properties verification is done on the models using the MDG tools. In both [13] and [2] inconsistencies in the assembly code with respect to the specification, as published in the application notes of the manufacturer, were uncovered through formal verification. In this paper we will investigate the verification of the same microcontroller application using a hierarchal approach based on Cadence SMV.

3 Cadence SMV Description

Cadence SMV is a formal verification system for hardware designs, based on a technique called symbolic model checking [8]. A formal verification system verifies that every possible behavior of the target system satisfies the specification. This is in contrast to a simulator, which can only verify the system's behavior for the particular vectors provided.

Cadence version of SMV uses Verilog hardware description language to express the system model. It supports CTL model checking [7]. A specification for SMV is a collection of properties. A property can be as simple as a statement that a particular pair of signals are never asserted at the same time, or it might state some complex relationship in the values or timing of the signals. Properties are specified in a notation called temporal logic. This allows concise specifications about temporal relationships between signals, and can be automatically verified.

SMV is quite effective in automatically verifying properties of combinational logic and interacting finite state machines. Sometimes, when the checking of a property fails, the tool will automatically produce a counter-example. This is a behavioral trace of the finite state machines that violates the specified property. Thus making SMV a very effective debugging tool, as well as a formal verification system.

For large designs, especially those including substantial datapath components, the user must break the correctness proof down into small enough pieces for SMV to verify. There are two mechanisms provided for this purpose: *composition* and *refinement*. In the compositional method, one verifies temporal logic properties of one part of the system and uses these properties as assumptions when verifying another part of the system. In the refinement method, one uses a high level model of the system as a specification, and verifies separately that each system component implements its part of the high level specification.

A specification is a description of the intended required behavior of a hardware design [4]. Various formalisms have been used to represent specifications. SMV uses logic-based specification like modal logic (e.g. temporal logic, extended temporal logic). Other tools use automaton-based specification.

An implementation description for a task at any given level serves also as a statement of the specification for a task at the next lower level. In this manner top level specifications can be successively implemented and verified at each level, thus leading to the implementation of an overall verified system. By breaking a large problem into smaller pieces that can be handled individually, the verification problem is made manageable. It effectively increases the range of circuit sizes that can be handled in practice.

4 Microcontroller and Embedded Software Description

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: Control, Button detection, Motion detection, Interface signal generation (typically, RS-232), and DC power supply (Figure 1). 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 [9].

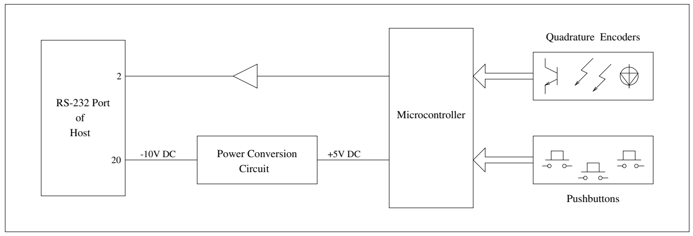

Figure 1: Functional blocks of a serial mouse

4.1 PIC16C71 Microcontroller

The microcontroller under verification is PIC16C71 [10], commercialized by Microchip Technology Inc. It is a RISC-like processor having only 35 instructions.The PIC16C71 is an 8-bit

controller, employing a RISC-like architecture Figure 2. There are 36 8-bit wide general purpose registers, a hardware stack and 15 special function registers (status register, low order 8-bit of the program counter (*pc*), 8-bit real time clock counter, etc.) [9]. The hardware stack is 8-level deep and has 36 bytes of RAM. A total of 35 instructions (reduced instruction set) are available, each instruction being 14-bit wide. A 1K EPROM memory contains the 14-bit instructions which compose the program. Each instruction takes one clock cycle to be executed, except for program branches which takes two cycles. An instruction cycle consists of eight Q cycles (Q1 to Q8). The instruction is fetched from the program memory and latched into the instruction register in Q3. This instruction is then decoded and executed during Q4, Q5, Q6, Q7, and Q8 cycles. Data memory is read during Q6 (operand read) and written during Q7 (destination write). A execution cycle ends with the *pc* incrementing in Q8.

4.2 Embedded Software

The embedded software under verification is the mouse controller software application written in PIC16C71 assembly language [9]. 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
- Subroutine *Byte*
- Subroutine *Bit*

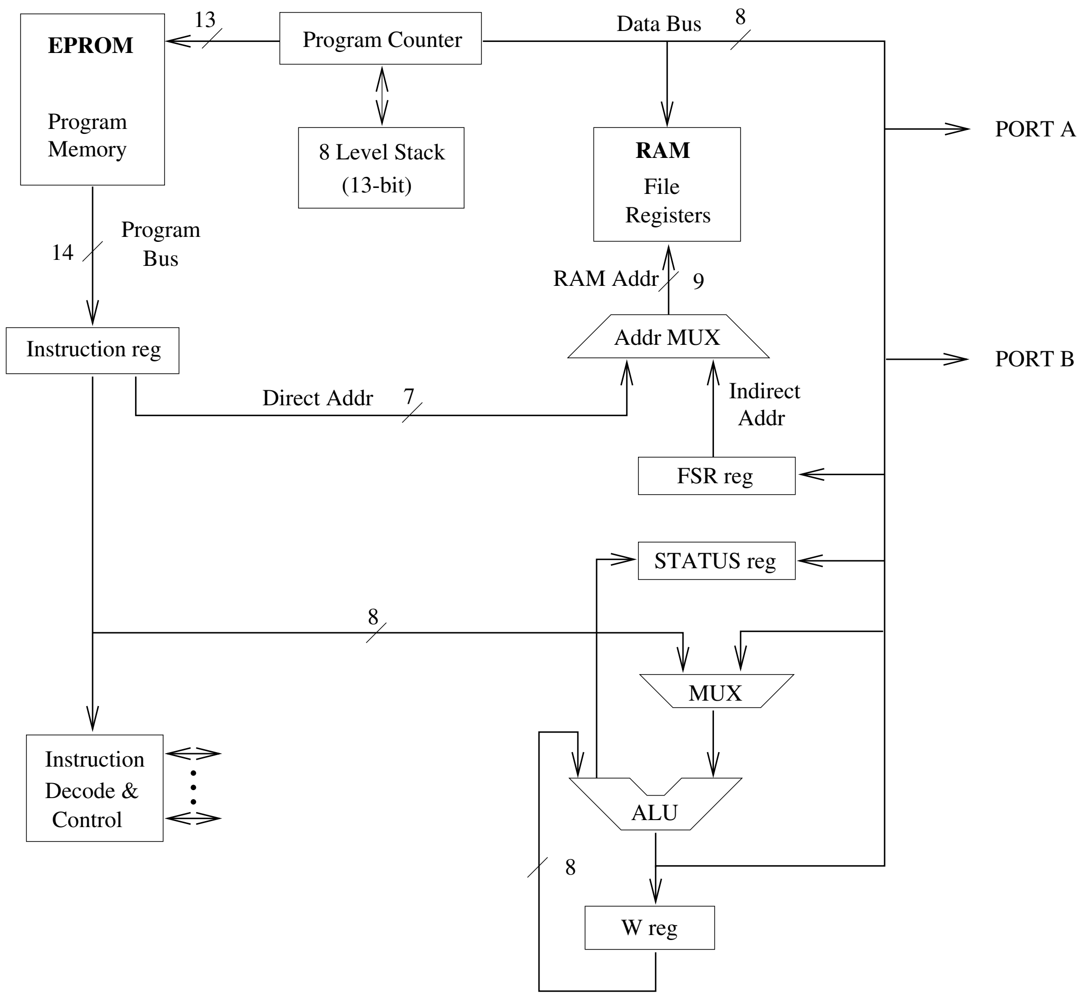

Figure 2: PIC16C71 RTL Block Diagram

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 description of the *Bit* routine is given below in Figure 3. 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 cleared *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 the *UpFlag*.

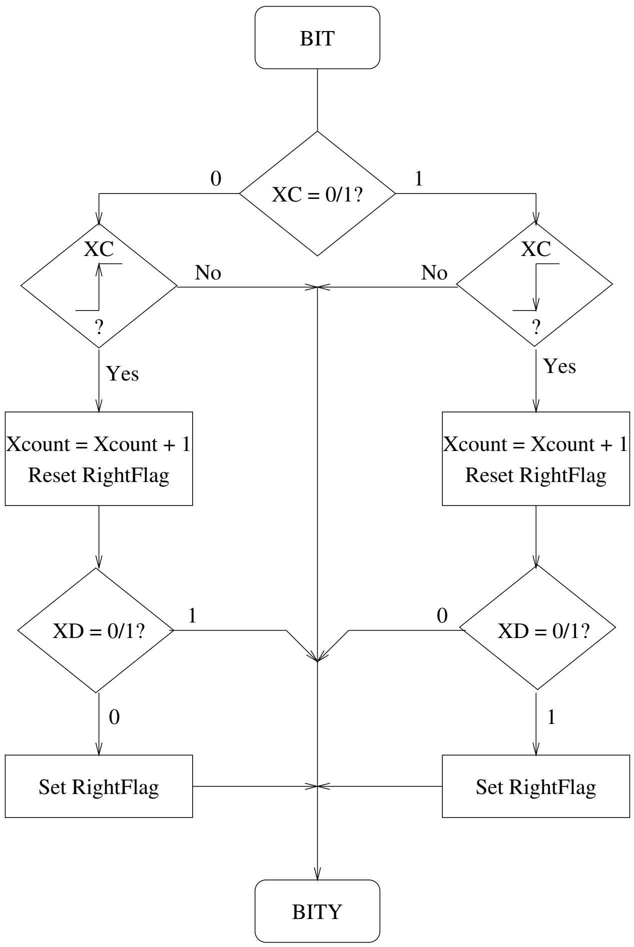

Figure 3: Flowchart of *Bitx* of Routine *Bit*

5 Verilog Modeling

The modeling language used by the Cadence SMV tool is Verilog. Each process is executed based on the instruction description in [9]. Each instruction in the routine is represented as a process in Verilog. The operation of each instruction is defined using the *module* definition in Verilog. SMV allows parallel processing. In order to run through each instruction cycle individually, i.e. manipulating the *pc*, each process carries an instruction number (*inr*) and a *pc*. Also the manipulation of the *pc* allows us to check the status of properties during each instruction cycle. The *pc* value, instruction number, source operand and the destination operand are passed as parameters to the modules. The eight Q cycles in an instruction are described using CASE statements inside each module.

We first implemented the model of the routine *Bit* at the RT level. The width of registers were taken according to the given PIC16C71X data sheet [10]. For illustration purposes, the opcode declared in main module is shown below.

```
INST0: process BTFSS (0,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,RA[2]);
```

INST0 is the label for the first instruction to be executed followed by *INST1*, *INST2* etc. *EP_out* is the 14-bit EPROM that stores the opcode. This is followed by the registers *IR_out,* and *Decode_out* that stores and decodes the current instruction to be executed respectively. Each instruction also flows through a bus *data_bus* and an ALU output, *alu_output*.

The opcode *BSF reg[b]* sets bit *b* of the register *reg*. *Process* is the keyword used to declare an instruction in the main module. The complete definition (function) of the above given declaration in main module is as follows:

```
module BSF (Inr, pc, Q, EP_out, IR_out,
Decode_out,data_bus, alu_output, flag){
     case {
          Inr= pc & Q= 1:{
               next(Q):= Q+1;
          }
          Inr= pc & Q= 2:{
               next(EP_out):= EP_out;
               next(Q):= Q+1;
          }
          Inr= pc & Q= 3:{
               next(IR_out):= EP_out;
               next(Q):= Q+1;
          }
          Inr= pc & Q= 4:{
               next(Decode_out):= IR_out;
               next(Q):= Q+1;
          }

          Inr= pc & Q= 5:{
               next(data_bus[2]):= flag;
               next(Q):= Q+1;
          }
          Inr= pc & Q= 6:{
               next(alu_output):= Decode_out;
               next(Q):= Q+1;
          }
          Inr= pc & Q= 7:{
               next(flag):= 1;
               next(Q):= Q+1;
          }
          Inr= pc & Q= 8:{
               next(pc):= pc +1;
               next(Q):= 1;
          }
     }
  }
```

At a higher abstraction level, we use process to model assembly programs. For example the *Bit1* of *Bitx* routine contains these instructions:

```
INST0: process BTFSS (0,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,RA[2]);
INST1 : process GOTO  (1,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,BIT0 );
INST2 : process BTFSC (2,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,CSTAT[2]);
INST3 : process GOTO  (3,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,BITY);
INST4 : process INCF  (4, pc, Q, EP_out, IR_out, Decode_out, data_bus, alu_output, W, XCOUNT, 1, STATUS[2]);
INST5 : process BCF(5,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,FLAGB[3]);
INST6 : process BTFSS (6, pc, Q, EP_out, IR_out, Decode_out, data_bus, alu_output, RA[3]);
INST7 : process GOTO  (7,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,BITY);
INST8 : process BSF(8, pc, Q, EP_out, IR_out, Decode_out, data_bus, alu_output, FLAGB[3]);
INST9 : process GOTO  (9,pc,Q,EP_out,IR_out,Decode_out,data_bus,alu_output,BITY );
```

The instructions are executed in sequence. The first three instructions detects the rising edge of clock cycle. Instruction #4 increments *XCount*, and instruction #5 resets the right flag during the rising edge of the clock. The rest of the instructions check if *XData* is set and then set the right flag.

6 Verification using SMV

SMV uses OBDD algorithm to check whether the CTL specifications are met [8]. We verify the liveness of the given model using this tool. The following is an example of three CTL properties we checked against *Bit1* model:

Property1: `assert G (( RA[2] =1 & CSTAT[2]=0) -> F (XCOUNT =1));`

Property2: `assert G (( RA[2] =1 & CSTAT[2]=0) -> F (FLAGB[3] =0));`

Property3: `assert G (( RA[2] =1 & CSTAT[2]=0 & XDATA=0) -> F (FLAGB[3] =1));`

The *assert* statements specify a number of properties that we would like to prove about this model. Note that *&* stands for logical “and” while -> stands for “implies”. In addition, the *F* operator is used to express a condition that must hold true at some time in the future. The formula *F p* is true at a given time if p is true at some later time. On the other hand, *G p* means that *p* is true at all times. Usually, we read *F p* as “eventually p” and *G p* as “henceforth p” [8].

Property1 describes that during the rising edge of the clock cycle (RA[2] =1 and CSTAT[2] =0) the X movement counter (XCOUNT) is incremented by 1 in all cases in the future. With *property 2* we can check that during the rising edge of the clock cycle (RA[2] =1 and CSTAT[2] =0) the general purpose flag (FLAGB) is reset. *Property 3* asserts that if XDATA =0 is read during a rising edge of the clock (RA[2] =1 and CSTAT[2] =0) then the right flag (FLAG[3] is set to 1 in all cases in the future [13]. Table 1 shows the model checking results of verification of *Bit1* of *Bitx* routine using Cadence SMV. This and subsequent experiments are done on a Sun Sparc Ultra1 with 256 MB memory.

Table 1: SMV model checking experimental results on original software

Properties	Verification	CPU time (sec)	BDD Nodes
property1	succeeded	2.45	65721
property2	succeeded	2.67	65721
property3	failed	4.79	77662

Through our experiment, we could notice that the test made over the embedded software written over the piece of micro controller device failed (Table 1). A counterexample by Cadence SMV notified the error. The error was traced back to instruction #6 which was found to be erroneous. The operation of the instruction is to be BTFSC instead of BTFSS. The error was subsequently corrected, and the experiment is conducted again. All the properties succeeded on the corrected software (Table 2).

Table 2: SMV model checking experimental results on corrected software

Properties	Verification	CPU time (sec)	BDD Nodes
property1	succeeded	2.45	65721
property2	succeeded	2.67	65721
property3	succeeded	2.84	62821

As further experiment, we conducted model checking of *property3* on increasing set of instruction processes, i.e., ascendingly including each of the four subroutines in the Bit routine (Figure 3). The results are plotted in Figure 4 which represents a linear variation between the two co-ordinates, the CPU time and the number of instructions. Table 3 presents the variation in the verification of this property including CPU time and number of BDD nodes allocated. We have found out that Cadence SMV is effective model checking tool especially if we are able to break down a complex design into smaller pieces is needed.

Table 3: Variation in property3 verification with increasing instruction processes

Number of process	CPU time (sec)	Nodes allocated
10 (*Bit0* of *BitX*)	2.84	62821
17 (*BitX*)	4.78	98507
27 (BitX & *Bit1* of *BitY*)	9.91	199566
34 (*BitX & BitY*)	11.83	238254

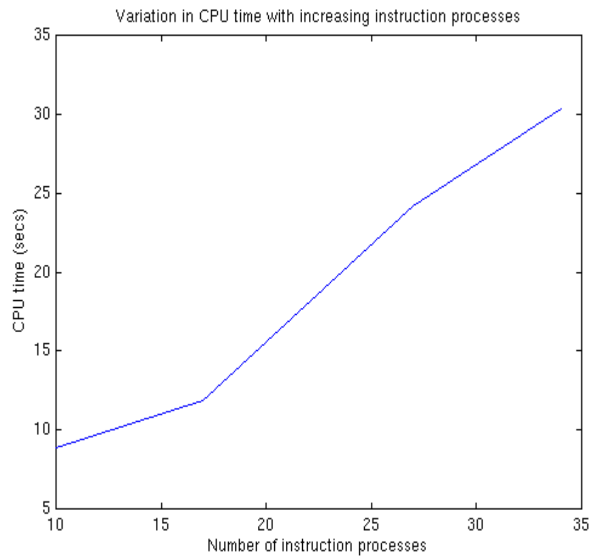

Figure 4: Variation in CPU time with increasing instruction process in checking property3.

Comparison with MDG

An experiment is conducted to have a comparison with the MDG model checking experiments in terms of time and complexity.

The properties mentioned earlier in Cadence SMV, expressed in L_{MDG} [14] are as follows:

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

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

AG means for all states in all parts. *X* means in next state. Note that *Property A* is equivalent to *property1* of Table 2 & *Property B* is similar to the CTL property verified in [13]. Table 4 summarizes the performance statistics of the model checking experiments on the specification and the implementation of the routine *Bit1* of *Bitx*.The experiments were conducted on SUN SPARC ULTRA1 with 256 MB of main memory.

Table 4: Performance statistics of model checking using MDG tools

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

In this paper, there is only one model, which is the model of the software routine running on the hardware. Even though, in [2] there are two models, one of the specification and the other of the implementation, of the software routine, comparing the results obtained in Table 1 with that in Table 4, show a remarkable reduction in the size of the graphs and the CPU time. Further more, Thiry and Claesen [13] report the verification of *Property A* on the *Bit* routine, run on a 486DX33 machine with 16 MB RAM. The verification time reported was 23 seconds. Table 4 shows that the properties verified on the *Bit* routine consumed less than a second of CPU time. This demonstrates the effectiveness in using MDG tools for model checking.

7 Conclusion

Commercial pressure to produce higher quality hardware and software is always increasing. Formal methods have already demonstrated success in specifying commercial and safety-critical application software; and in verifying protocol standards and hardware designs. Progress, however will depend on continuing support for basic research on new specification languages and new verification techniques. The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last ten years.

In this report, we present a methodology and application of the formal verification of embedded software using the SMV tool. We model the instruction set of a commercial microcontroller in the Cadence SMV input language, Verilog. We work out a method for the symbolic verification of the embedded software, in terms of assembly code, running on the above mentioned processor. We use the application of a mouse controller software with Microsoft compatible RS232 interface as an example to demonstrate our approach. Throughout our experiment, we uncovered inconsistencies between the specification and the implementation in the assembly code.

References

[1] S. Balakrishnan and 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, Machigan, USA, March 1999, pp. 284-287.

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

[3] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny: Multiway Decision Graphs for Automated Hardware Verification; *Formal Methods in System Design*, Vol. 10, February 1997, pp. 7-46.

[4] R. Hojati, R. K. Brayton. Automatic Datapath Abstraction in Hardware Systems. *Computer Aided Verification*, LNCS 939, Springer Verlag, 1995, pp. 98-113.

[5] C. Kern and M. Greenstreet: Formal Verification in Hardware Design: A Survey; *ACM Transactions on Design Automation of E. Systems*, Vol. 4, April 1999, pp. 123-193.

[6] R. P. Kurshan: Formal Verification in a Commercial Setting; *Proc. Design Automation Conference*, Anaheim, California, June 9-13, 1997, pp 258-262.

[7] K. McMillan: *Symbolic Model Checking*; Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[8] K. McMillan: *Getting started with SMV*; User's Manual, Cadence Berkeley Laboratories, USA, 1998.

[9] Microchip Technology Inc.: *Embedded Control Handbook*; 1993, pp. 2. 121-2.133.

[10] Microchip Technology Inc.: *PIC16C71*; 1994, pp. 2.328-2.372.

[11] C. Seger: An Introduction to Formal Verification; *Technical Report 92-13*, UBC, Department of Computer Science, Vancouver, B.C., Canada, June 1992.

[12] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, and O. Ait-Mohamed: Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs; *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 7, July 1999, pp. 956-972.

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

[14] Y. Xu. MDG Model Checker User's Manual, Dept. D'IRO, University of Montreal, Montreal, September 1999.