On the Formal Verification of Embedded Software Using Multiway Decision Graphs

Subhashini Balakrishnan and Sofiène Tahar

Technical Report No 402

December 1997

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

Abstract. *Embedded systems are finding widespread application including communication systems, factory automation, graphics and imaging systems, medical equipment and even household appliances. We endeavor to highlight the need for formally verifying the correctness of an embedded software, and suggest a methodology for the same using Multiway Decision Graphs (MDG). This is particularly necessary for real-time and safety critical applications. We propose a formal model for the Instruction Set Architecture, along with the underlying hardware (RT level) of a microcontroller in commercial use. The embedded software we consider is used in a mouse controller application. Models for the flowchart specification and the assembly language implementation of the embedded software are also portrayed. Using the MDG verification tools, equivalence checking experiments are conducted between the flowchart specification of the embedded software and its assembly code implementation. In addition, specific properties of the mouse controller behavior are also checked on all our models. Inconsistencies in the assembly code with respect to the specification, as published in the application notes of the manufacturer, were uncovered through both equivalence and property checking. Given the relatively small CPU time and memory consumption achieved in our experiments, the verification methodology we adopted would be able to verify significantly larger programs using the MDG tools and can hence cope with the inherent complexities of verifying a whole embedded system.*

Introduction

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. Interest in hardware/software codesign [2] 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. Recently attention has been given to the verification of embedded systems using formal methods.

In this paper we present a methodology and application of formal verification of embedded software, using Abstract State Machines (ASMs) [5] based on Multiway Decision Graphs (MDGs) [5]. We demonstrate our approach on the embedded software used to program the microcontroller PIC16C71, commercialized by Microchip Technology Inc., [8] for a serial mouse controller application [7].

Some work have been done on the formal verification of the hardware of microprocessors and digital signal processor algorithms. These include theorem prover methodologies [9, 11] or Decision Diagram based approaches [3, 5]. Thiry and Claesen [10] suggested a methodology for modeling both the software and hardware of an embedded system using the CTL temporal logic representation, implemented in SMV [6]. They presented a way for checking properties on the embedded software. SMV uses the ROBDD symbolic model checking algorithm to find out whether CTL specifications are satisfied. More recently, in [1] the verification of an assembly code for the Motorola Complex Arithmetic Digital Signal Processor using ACL2 theorem-proving system is described. Unlike model checkers, theorem provers are scalable to large software programs but are interactive. Our work is motivated by the fact that a symbolic model checker is restricted to representation at the boolean level and a theorem prover to users with a lot of expertise and experience. We illustrate the ability to carry out equivalence checking, in addition to checking properties, using ASMs. We thus pave a way for automatic verification of a complete embedded software program and eventually a complete embedded system at higher levels of abstraction.

The rest of this paper is organized as follows: Section 1 is a brief introduction to Multiway Decision Graphs and its related verification techniques. The architecture and the implementation of the microcontroller and their models are presented in Section 2. Section 3.2 describes the mouse controller function and the related MDG models for the embedded software. Section 4 outlines the verifications we carried out using our models and Section 5 summarizes the experimental results. We reach our conclusions and ideas on further work in Section 6.

1 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) have been proposed recently [5] as a solution to the state space explosion problem of ROBDD based verification tools. 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. It is based on a subset of many-sorted first order logic, augmented with a distinction between *abstract* and *concrete* sorts. Concrete sorts have *enumerations* which are sets of *individual constants*, while abstract sorts do not. The enumeration of a concrete sort is a set of distinct constants of that sort. The constants occurring in enumerations are referred to as *individual constants*, and other constants as *generic constants*.

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 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 the MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs incorporate variables of abstract type to denote data signals and *uninterpreted function* symbols to denote data operations. MDGs can also represent sets of states. 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.

In MDG-based verification, abstract descriptions of state machines, called *abstract state machines* (ASM) [5] 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. 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.

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 [5]. The MDG tools package the basic MDG operators and verification procedures. The operators are disjunction, pruning-by-subsumption, and term-rewriting. The verification procedures are combinational and sequential verification. The combinational verification provides the equivalence checking of two combinational circuits. The sequential verification provides invariant checking and equivalence checking of two state machines. The MDG tools run on a Prolog platform and they accept a Prolog-style HDL, MDG-HDL [12], 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.). 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. Interested readers are referred to [4, 5, 12] for more details on MDG and the tools.

2 The Microcontroller Architecture

The hardware device we investigate in this work is the PIC16C71 microcontroller [8], commercialized by Microchip Technology Inc. We give a simple, brief description of the microcontroller architecture and implementation, with their MDG models.

2.1 MDG Modeling of the RTL Implementation

The PIC16C71 is an 8-bit controller, employing a RISC-like architecture (Figure 1). There are 36 8-bit wide general purpose registers, 15 special function registers and a hardware stack. 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 two stage instruction pipeline allows all instructions to execute in a single clock cycle, except for program branches, which require two cycles.

Using MDG-HDL, we described the RTL netlist implementation of the microcontroller of Figure 1. We adopted a hierarchical description down to the MDG-HDL library of basic components (see Section 1). As the MDG system can handle abstract descriptions, it 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}. In MDG-HDL, the system bus can be modeled using the basic component, drivers. The program memory is described in terms of access functions e.g., *read*, *write*, *fetch* etc., modeled as uninterpreted function symbols. The ALU can be viewed as a black box, where ALU functions are expressed using uninterpreted function symbols e.g., *add*, *sub*, *inc*, *or* etc.

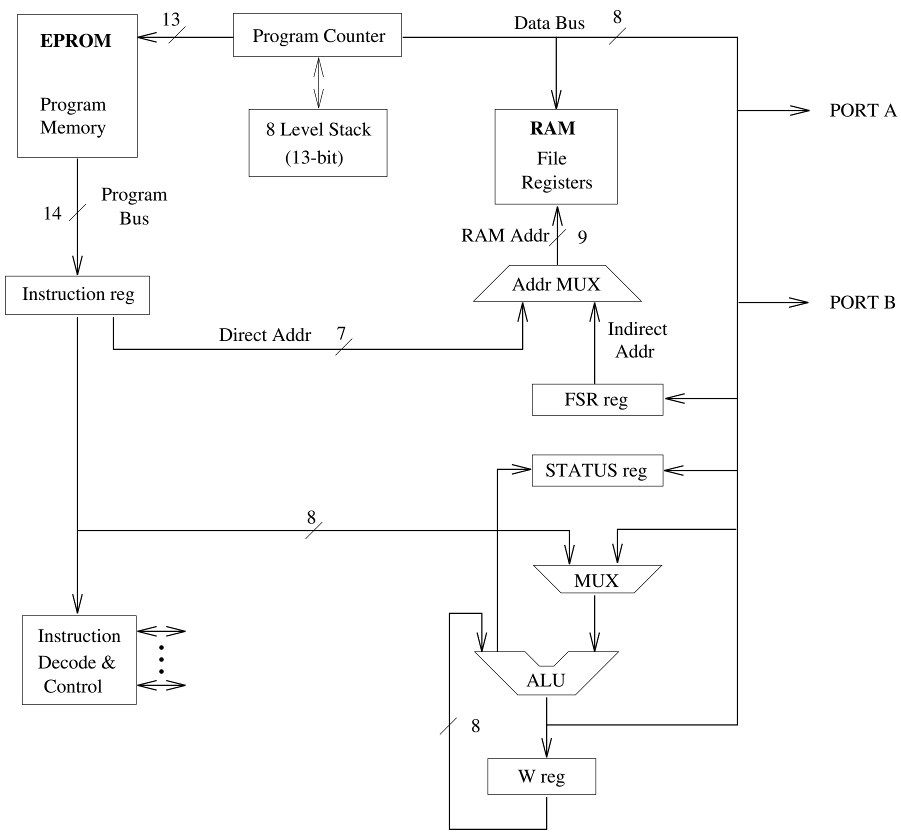


Figure 1: Microcontroller architecture (RT-level)

2.2 MDG Modeling of the ISA

The instruction set architecture (ISA) consists of byte-oriented, bit-oriented, literal and control operations. The following are the four general formats of instructions:

- instructions acting on two registers
- instructions acting on the working register, *w*
- instructions acting on one bit
- instructions acting on the program counter, *pc*

Using MDGs we can model the assembly instructions as predicates. We illustrate this on the following two instructions: the inclusive-OR instruction (IORLW K) [8] between a literal (k) and a working register (w), and the decrement instruction (DECF W) [8] of the working register (w). The operations (e.g., inclusive-OR, decrement) can be modeled using uninterpreted functions (e.g., *or*, *dec*), applied to the arguments of the instructions (predicates). The instructions mentioned above and their models, written in MDG-HDL, are as follows:

Assembly Instruction: `IORLW K`

MDG-HDL Model: Definitions:

```
var(w, worda8)
var(k, worda8)
function(or, inputs[worda8, worda8], output[worda8])
```

Instruction:

```
transform(inputs([w, k]), function(or), output(w))
```

Assembly Instruction: `DECF W`

MDG-HDL Model: Definitions: `var(w, worda8)`
`function(`**dec**`, input[worda8], output[worda8])`

Instruction: `transform(inputs([w]), function(`**dec**`), output(w))`

Besides the ISA, we also model the execution of the assembly instructions at the microprogram level. The microprogram counter, *mpc,* fetches the opcode from the program memory, and on decoding the opcode, controls the value of the *pc* and the contents of the registers. The *mpc* simulates the two cycles, fetch and execute of each instruction. In MDG-HDL, we model the *mpc* as a concrete variable of sort *bool*. We modeled the opcode fetch using an uninterpreted function such as *opcode(fetch(mem, address(pc)))* and decoding specific bits of an instruction can be modeled in a similar way, e.g., *src_op(fetch(mem, address(pc)))*.

3 Embedded Software

3.1 MDG Modeling of the Embedded Software

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 instructions in an assembly code or the statements of a specification control the transitions from one state to another. In the following we present a model for the behavioral level, its implementation in the assembly code level and its execution at the Register Transfer (RT) level of the embedded software [7] using MDGs.

3.1.1 Assembly Language Program Implementation

The implementation considered here is the assembly language code programmed (embedded) in a microcontroller. Figure 2 illustrates an example piece 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 2. The ASM model has five states, each state implementing one instruction of the program. Labels “00” to “04” refer to the five states of the ASM. In the MDG-HDL model the registers *R1* and *R2* are of abstract sort *word8*. The model uses the functions *dec* and *inc* to decrement and increment a register, respectively. They are defined with following types[1]:

dec: [*word8* $\longrightarrow$ *word8*]
inc: [*word8* $\longrightarrow$ *word8*]

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.

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

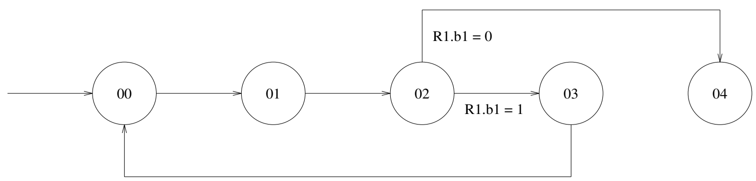
Assembly code:

```
00 START DECF R1
01       INCF R2
02       BTFSC R1.b1
03       GOTO START
04       .........
```

Notations:

```
DECF Ri     decrement and updates register Ri
INCF Ri     increment and updates register Ri
BTFSC Ri.bj test bit j of reg Ri. If cleared, skip next instruction
GOTO  L     jump to the address indicated by the label L
```

ASM model:



MDG-HDL model:

```
initialize pc = 0
case pc = 0 : next(pc) = 1
              R1 = dec(R1)
case pc = 1 : next(pc) = 2
              R2 = inc(R2)
case pc = 2 : if (R1.b1 = 0) then next(pc) = 4
                                  else next(pc) = 3
case pc = 3 : next(pc) = 0
```

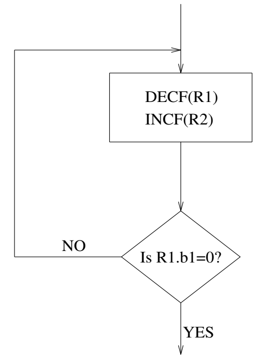
Figure 2: An example assembly code fragment and its MDG model

While, each instruction in the implementation (the embedded assembly language code) of the microcontroller must be executed in succession, the MDG system interprets its statements, written in MDG-HDL code, in parallel, unless otherwise stated. So, in order to suppress the inherent parallelism of the MDG system, the program counter, *pc*, is used as the sequencer (see Figure 2).

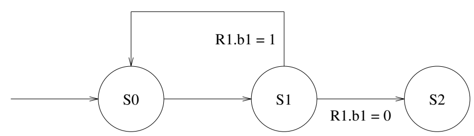
3.1.2 Algorithmic Flowchart Specification

Our main goal is to verify whether the programmed embedded software confirms with the intended behavior. One common way of describing the intended behavior is using algorithmic flowcharts. Hence, we consider the flowchart of the embedded software program as the behavioral specification. We model it as an ASM (given mainly by a tabular representation of the transition and output relations, and some predefined components in MDG-HDL). An example flowchart and its model in pseudo-MDG-HDL is shown in Figure 3. Since the specification is an abstract behavior, the sequential ASM behavior would take lesser clock cycles than the implementation. This is evident from the number of states in the state diagram in Figure 3 as compared to that in Figure 2. During the verification process, we introduce a synchronization flag in order to relate the two descriptions, as will be outlined in Section 4.1.

Flowchart:



ASM model:



MDG-HDL model:

```
initialize s = 0
case s = 0 :  R1 = dec(R1)
              R2 = inc(R2)
              next(s) = 1
case s = 1 :  if (R1.b1 = 0) then next(s) = 2
                                 else next(s) = 0
```

Figure 3: An example flowchart and its ASM model

3.2 The Mouse Controller Software

The mouse is becoming increasingly popular as a standard pointing data entry device. The intelligence of the mouse is provided by the microcontroller, therefore the features and performance of a mouse is greatly related to the microcontroller used. The implementation of a serial mouse using the PIC16C71 [7] is taken as the example for our purpose. We started with the data sheets of PIC16C71 by Microchip Technology Inc. [8].

The major tasks performed by the embedded software for the mouse controller are:

- Button scanning
- X and Y motion scanning
- 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*

The main program detects any changes in the button status and in the movement counts. There are two subroutines *Byte* and *Bit*. The *Byte* is called in the main program five times to send five bytes of data. The *Byte*, in turn calls the routine *Bit* periodically. The *Bit* counts the number of pulses from the outputs of the photo detectors and determines the direction of movement. In this paper, we concentrate on the specification of the routine *Bit* for our demonstration.

The routine *Bit* [7] 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*.

3.2.1 Algorithmic Flowchart Specification

Our main goal is to verify whether the programmed embedded software confirms with the intended behavior. One common way of describing the intended behavior is using algorithmic flowcharts. Hence, we consider the flowchart of the embedded software program as the behavioral specification. For the purpose of our demonstration, we consider the flowchart of the routine *Bit* and derive the specification for the *Bit1* section (Figure 4) of *Bitx* [7].

We model the flowchart as an ASM (given mainly by a tabular representation of the transition and output relations). 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. In our application, each state in the ASM characterizes the contents of the microcontroller registers. The instructions in an assembly code, or a microcode, or the statements of a specification control the transitions from one state to another.

The ASM model comprises of 2 states (S0 and S1), one for resetting the *Right Flag* and incrementing the *XCount*, and the other for setting the *Right Flag* (Figure 4). The model uses the abstract sorts *worda8* and *label* for the *XCount* and (branch) target address labels, respectively. We thus do abstract away from defining any concrete bit width for the *XCount* and any enumeration for the address labels. We also use two concrete sorts *bool* and *wordc4*. The former is the boolean sort while the latter designates 4 bit words with enumeration {0, ..., 15} and is used for *pc* values. Besides, the following functions are declared:

rise: [*bool* ⟶ *bool*]

reset: [*bool* ⟶ *bool*]

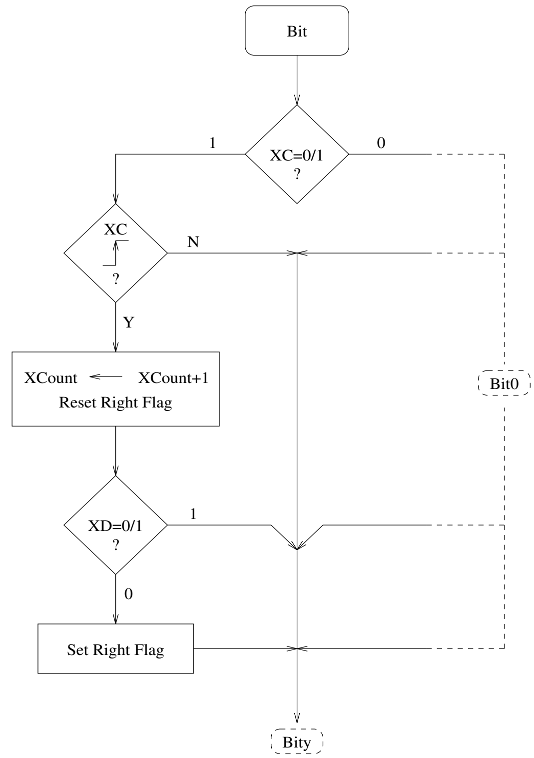
inc: [*worda8* ⟶ *worda8*]

set: [*bool* ⟶ *bool*]

goto: [*label* ⟶ *wordc4*]

The function *rise* detects the positive edge of *XC*, *inc* increments the *XCount*, *reset* and *set* resets and sets the *Right Flag* respectively, and the function *goto* transfers control to the *Bit0* or *Bity* sections as the case may be. The *Right Flag*, *XC* and *XD* are declared of concrete sort *bool, XCount* of abstract sort *wordn*, and *Bit0* and *Bit*y of abstract sort *label*.

Flowchart:



MDG-HDL model:

```
initialize s = 0
case s = 0: if (XC = 0)
               then if (rise(XC))
                       then inc(XCount)
                            reset(RightFlag)
                            next(s) = 1
                       else goto(BITY)
               else goto(BIT0)
case s = 1: if (XD = 0) then set(RightFlag)
                             goto(BITY)
```

Figure 4: Flowchart specification of *Bit1* routine and its MDG model

3.2.2 Assembly Language Program Implementation

The implementation considered here is the assembly code (instructions numbered 00 to 09) of the *Bit1* section of *Bitx* (Figure 5) programmed (embedded) in the microcontroller, along with its model in MDG-HDL. The positive edge of *XC* is detected by the bit 2 of register *RA* (*RA.b2*) being set, and bit 2 of register *CSTAT (CSTAT.b2)* being cleared. In the implementation, *XD* is implemented as bit 3 of *RA* (*RA.b*3), and the *Right Flag* as bit 3 of *FLAGB* (*FLAGB.b3*).

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 5. The MDG ASM model consists of ten states (S0 to S9), each for implementing one instruction at a time. The variables *RA.b2*, *CSTAT.b2*, *FLAGB.b3* are of sort *bool*, *pc* of sort *wordc4*, and *Bit0* and *Bity* of sort *label.* Furthermore, we use the same abstract and concrete functions as in the flowchart specification. This is mandatory when checking equivalence between the specification and the implementation using the MDG system.

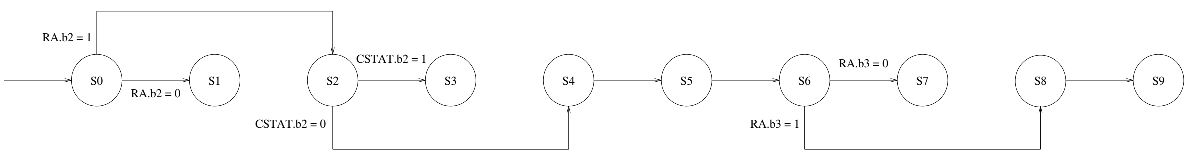
Assembly language program:

```
00: BTFSS RA.b2
01: GOTO BIT0
02: BTFSC CSTAT.b2
03: GOTO BITY
04: INCF XCOUNT
05: BCF FLAGB.b3
06: BTFSS RA.b3
07: GOTO BITY
08: BSF FLAGB.b3
09: 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 = 0:  if (RA.b2 = 0)then next(pc) = 1
                            else next(pc) = 2
case pc = 1:  goto(BIT0)
case pc = 2:  if (CSTAT.b2 = 1) then next(pc) = 3
                                 else next(pc) = 4
case pc = 3:  goto(BITY)
case pc = 4:  inc(XCOUNT)
              next(pc) = 5
case pc = 5:  reset(FLAGB.b3)
              next(pc) = 6
case pc = 6: if (RA.b3 = 0) then next(pc) = 7
                            else next(pc) = 8
case pc = 7:  goto(BITY)
case pc = 8:  set(FLAGB.b3)
              next(pc) = 9
case pc = 9:  goto(BITY)
```

Figure 5: Assembly code implementation of *Bit1* routine with its ASM and MDG model

While, each instruction in the implementation (the embedded assembly language code) of the microcontroller must be executed in succession, the MDG system interprets its statements, written in MDG-HDL code, in parallel, unless otherwise stated. So, in order to suppress the inherent parallelism of the MDG system, the program counter, *pc*, is used as the sequencer (see Figure 5).

Since the specification is an abstraction of the implementation, the sequential ASM behavior would take lesser clock cycles than its implementation. During the verification process, we introduce a synchronization flag in order to relate the two descriptions, as will be outlined in Section 4.1.

4 Formal Verification

4.1 Equivalence checking

Equivalence checking between the specification or the behavior and the implementation is done to ensure that the implemented 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. While a specification does not necessarily produce its outputs during the same clock cycles as its implementation does, we need a way to synchronize both. We achieve this by introducing a variable, *sync*, which keeps track of the sequence of each instruction in the assembly code implementation. This is passed as an input to the specification, wherein it delays each of its output until its value equals that of the *pc* of the instruction in the implementation that produces the same output.

Below is shown the synchronized MDG model for the specification that is amenable to sequential equivalence checking. In the implementation (both the assembly and the microprogram level) we declare the new variable *sync* to be of sort *wordc4*. We define it equal to the *pc*. It is given as an output from the implementation and is taken as an input by the specification. The introduction of the *when* statements in the specification is done in a simple way in MDG_HDL by adding a new input column in the table description under the name *sync*.

Synchronized MDG-HDL model for specification:

```
initialize s = 0
case s = 0: if (XC = 0)
              then if (rise(XC))
                     then when (sync = 4) inc(XCount)
                          when (sync = 5) reset(RightFlag)
                          next(s) = 1
                     else when (sync = 3) goto(BITY)
              else when (sync = 1) goto(BIT0)
case s = 1: if (XD = 0)
              then when (sync = 8) set(RightFlag)
                   when (sync = 9) goto(BITY)
              else when (sync = 7) goto(BITY)
```

4.2 Property checking

Property checking is used for verifying that a design satisfies some specific requirements. MDG tools allow checking of safety properties as invariants. Therefore, apart from conducting equivalence checking, we also checked some properties of the *Right Flag* at particular time points on all our models.

Following are two example properties that test the status of the *Right Flag* in accordance with the primary inputs:

Property 1: **If** RA.b2 ∧ [¬ CSTAT.b2] **then** [¬ RightFlag]

Property 2: **If** RA.b2 ∧ [¬ CSTAT.b2] ∧ [¬ XDATA] **then** RightFlag

Property 1 checks whether the *Right Flag* is reset, when the inputs *RA.b2* and *CSTAT.b2* are 1 and 0 respectively. This property is checked at state 6 on the assembly implementation (Fig. 6 shows the state machine corresponding to the input conditions) or state 0 on the specification. Property 2 checks whether the *Right Flag* is set, when the inputs *RA.b2, CSTAT.b2* and *XDATA* are 1, 0 and 0 respectively. This is checked at state 9 on the assembly implementation or state 1 on the specification. This property is similar to the CTL property verified in [10].

We first transform the If...statements and the then....statements, each into a *Directed Formula* (DF) [4]. 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 statement is compared with the output of the circuit corresponding to the then statements. The equivalence between the two is checked as an invariant.

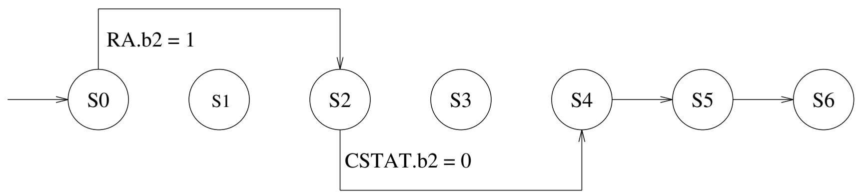


Figure 6: State machine corresponding to inputs RA.b2 = 1, CSTAT.b2 = 0

5 Experimental Results

Applying our models we verified the embedded assembly language software against its behavioral specification. We also validated our models and methodology by verifying the properties described in Section 4.2. The experiments were conducted on a SUN SPARC ULTRA 1 with 124 MB of main memory.

5.1 Equivalence checking

The equivalence checking between the behavioral specification and the embedded software 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. Accordingly, the instruction 6 is to be BTFSC in place of BTFSS (Figure 5). This result confirms with the one obtained in [10] using SMV. The erroneous instruction is corrected and the equivalence checking is run successfully. The performance statistics for the equivalence checking are given in Table 1, including CPU time, memory usage and number of MDG nodes generated.

Table 1: Performance statistics of equivalence checking

Verification on	Result of Verification	CPU time (seconds)	Counterex. gen. (seconds)	Memory usage (MB)	No. of MDG Nodes
original assembly impl.	*failed*	3.87	0.52	1.88	1836
corrected assembly impl.	*successful*	3.71	-	1.50	1555

5.2 Property checking

The properties stated in Section 4.2 of the *Right Flag* are checked on the flowchart (behavior) specification as well as on the embedded assembly language implementation. The property 1 succeeded on all the above mentioned state machines. This implies that an error, if any, could be occurring after state 6 of the implementation, at which property 1 is checked. Property 2 fails to succeed on the assembly language implementation. The MDG tool generates a counterexample, which confirms with the error located by the equivalence checking experiments. The instruction 6 is corrected and the properties 1 and 2 are checked to hold good on the corrected implementation machine. The performance statistics for the property checking are given in Table 2, including CPU time, memory usage and number of MDG nodes generated. Environment state machine for property checking

Table 2: Performance statistics of property checking

Property	Verification on	Result of Verification	CPU time (seconds)	Counterex. gen. (seconds)	Memory usage (MB)	No. of MDG Nodes
Property 1	original assembly implementation	*successful*	1.37	-	1.08	592
Property 1	corrected assembly implementation	*successful*	1.34	-	1.06	567
Property 1	specification	*successful*	1.31	-	1.06	567
Property 2	original assembly implementation	*failed*	1.44	0.10	1.10	711
Property 2	corrected assembly implementation	*successful*	1.37	-	1.08	613
Property 2	specification	*successful*	1.46	-	1.07	588

6 Conclusions

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 paper attempts to emphasize the imperative need for verifying the correctness of the software embedded in such a system. Models are outlined for the hardware of a a commercial microcontroller using MDGs. MDG models for expressing the behavioral specification and the implementation in the assembly code level of its mouse controller application are also established. The verification of the *Bit1* section of the mouse controller software for its correctness is demonstrated using the MDG tools. The experiments concluded in few seconds of CPU time.

Although our example software is small, it is ideal for illustration purposes as it covers the most recent applications of the MDG system. But, it does not represent the scale of programs we can verify. Given the relatively small CPU time and memory consumption achieved in our experiments, the verification methodology we adopted would be able to verify significantly larger pro grams using the MDG tools and can hence cope with the inherent complexities of verifying a whole embedded system. Checking of liveness properties would also be made possible by incorporating the recently developed Abstract Computational Tree Logic (Abstract-CTL) within the MDG system. Work is also underway in representing our models in standard hardware description languages, namely VHDL, which has been recently interfaced to the MDG tools. All these features would further enhance the applicability of the MDG tools for the verification of embedded systems.

Acknowledgments

We would like to express our gratitude to the MDG group at University of Montreal, especially Dr. Otmane Ait-Mohamed, for their interest and guidance. Our thanks are also due to Prof. Luc Claesen, IMEC/K. U. Leuven, Belgium, for his valuable suggestions on the script.

References

[1] B.C. Brock and 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, Texas, USA, October 1997, pp. 31-36.

[2] 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.

[3] J. Burch and D. Dill. Automatic Verification of Pipelined Microprocessor Control, Computer Aided Verification, Lecture Notes in Computer Science 818, springer Verlag, 1994, pp. 68-80.

[4] 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.

[5] 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, No. 1, 1997, pp. 7-46.

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

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

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

[9] S. Tahar, and 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.

[10] 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.

[11] P.Windley. Formal Modeling and Verification of Microprocessors. IEEE Transactions on Computers, Vol. 44, No. 1, 1995, pp. 57-72.

[12] Z. Zhou and N. Boulerice. MDG Tools (V1.0) User's Manual, Dept. D'IRO, University of Montreal, 1996.