HW/SW Co-Verification of a RISC CPU using Bounded Model Checking

Daniel Große Ulrich Kühne Rolf Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{grosse, ulrichk, drechsle}@informatik.uni-bremen.de

Abstract

Today, the underlying hardware of embedded systems is often verified successfully. In this context formal verification techniques allow to prove the correctness. But in embedded system design the integration of software components becomes more and more important. In this paper we present an integrated approach for formal verification of hardware and software. The approach is demonstrated on a RISC CPU. The verification is based on bounded model checking. Besides correctness proofs of the underlying hardware the hardware/software interface and programs using this interface can be formally verified.

1. Introduction

In the last few years embedded system design has become a very important research area and the application domains range from telecommunication devices to automotive components. These systems not only consist of hardware components, i.e. a large portion is realized by firmware and programs. Since these systems are used more and more in safety critical applications, the aspect of verification is very important to ensure the correct functional behavior of the system.

In the meantime hardware verification has been intensively studied and is well understood, even though the tools sometimes suffer from limit of resources. But assertion-based verification and formal approaches have ensured high quality also for large hardware systems. This standard is not achieved so far, if software components are included. E.g. a recent study by Collett International Research Inc. has shown that errors caused by firmware and hardware/software interfaces account for up to 13 percent of failures with an increasing trend. To reduce this type of errors in embedded systems integrated hardware/software verification is needed.

A successful technique for verification of hardware is *Bounded Model Checking* (BMC) [3]. BMC checks whether a circuit satisfies a temporal property or not. Therefore, BMC reduces the problem to a Boolean satisfiability problem and searches for counter-examples in executions whose length is bounded by k time steps.

In this paper we show that the concepts of BMC can also be applied in the context of hardware/software integration. For a RISC CPU it is demonstrated that a complete verification can be performed that includes the underlying hardware, the single assembler commands and even programs based on sequences of commands.

To provide some more details on the proof processes in the following we briefly review some of the notation and formalism. We make use of BMC as described in [10], i.e. a property only argues over a finite time interval and during the proof there is no restriction to reachable states. The general structure of the resulting BMC instance for a property p over the finite interval $[0, c]$ is given by:

$$\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \ \wedge \neg\, p$$

where $T_\delta(s_i, s_{i+1})$ denotes the transition relation between cycles i and $i+1$. Typically the property p consists of two parts: an *assume part* which should imply the *proof part*, i.e. if all assumptions hold, all commitments in the proof part have to hold as well.

Example 1. *A simple example formulated in PSL [1] is shown in Figure 1. The property* `test` *says that whenever signal* x *becomes* 1*, two clock cycles later signal* y *has to be* 2*.*

The integrated verification approach that supports hardware and software is based on the unrolling process of BMC. This also enables to consider sequences of instructions of the RISC CPU. The steps for this verification are explained in the following.

```
property test
always
 // assume part
 ( x = 1 )
 ->
 // prove part
 next[2] ( y = 2 );
```

Figure 1. Property `test`

The verification of the underlying *hardware* is done by classical application of BMC. At this stage all hardware units are formally verified by describing their behavior with temporal properties. This guarantees the functional correctness of each hardware block. Basically applying BMC for this task corresponds to block-level verification.

The *interface* is viewed as a specification that exits between hardware and software. By calling instructions of the interface an interaction with the underlying hardware is realized. At the interface the functionality of the hardware is available but the concrete hardware realization is abstracted. In contrast to block-level verification the interface verification with BMC formulates for each interface instruction the exact response of all hardware blocks involved. Besides these blocks it is also assured that no side effects occur.

Based on instructions available through the interface a *program* is a structural sequence of instructions. By a combination of BMC and inductive proofs [4, 8] a concrete program can be formally verified. Arguing over the behavior of a program is possible by constraining the considered sequence of instructions as assumptions in a BMC property. Thus, the property checker "executes" the program and can check the intended behavior of the proof part. Inductive reasoning is used to verify properties which describe functionality where the upper time bound varies, e.g. this is the case if loops are used.

2. Case Study: RISC CPU

This section provides the basics of the RISC CPU and details on the SystemC model. Then the verification process for hardware and software is described.

2.1. Specification

In Figure 2 the main components of the RISC CPU are shown. The CPU has been designed as a Harvard architecture. The data width of the program memory and the data memory is 16 bit and the sizes are 4 KByte and 128 KByte, respectively. The length of an instruction is 16 Bit. In the following we only briefly describe the five different classes of instructions of the RISC CPU:

- 6 load/store instructions (movement of data between register bank and data memory or I/O device, load of a constant into high- and low byte of register, respectively)
- 8 arithmetic instructions (addition/subtraction with and without carry, left/right rotation and shift)
- 8 logic instructions (bit by bit negation, bit by bit exor, conjunction/disjunction of two operands, masking, inverting, clearing and setting of single bits of an operand)
- 5 jump instructions (unconditional jump, conditional jump, jump on set/cleared carry or zero flag)
- 5 other instructions (stack instructions push and pop, program halt, subroutine call, return from subroutine)

For more details on the CPU we refer the reader to [2].

2.2. SystemC Model

The RISC CPU has been modeled in the system description language SystemC [9, 7]. As a C++ class library SystemC enables modeling of systems at different levels of abstraction starting at the functional level and ending at a cycle-accurate model. The well-known concept of hierarchical descriptions of systems is transferred to SystemC by describing a module as a C++ class. Furthermore, fast simulation is possible at an early stage of the design process and hardware/software co-design can be carried out in the same environment. Note that a SystemC description can be compiled with a standard C++ compiler to produce an executable specification.

For details on the SystemC model of the RISC CPU we refer the reader to [6]. To simulate RISC CPU programs a compiler has been written which generates object code out of the assembler language of the RISC CPU. This object code runs on the SystemC model, i.e. the model of the CPU executes an assembler program. During the simulation tracing of signals and states of the CPU is possible.

2.3. Formal Verification

For property checking of the SystemC model the tool presented in [5] is used. It is based on SAT solving techniques and for debugging a waveform is generated in case of a counter-example. In the following the complete verification of the hardware, interface and assembler programs for the RISC CPU is discussed. All experiments have been carried out on a Athlon XP 2800 with 1 GByte of main memory.

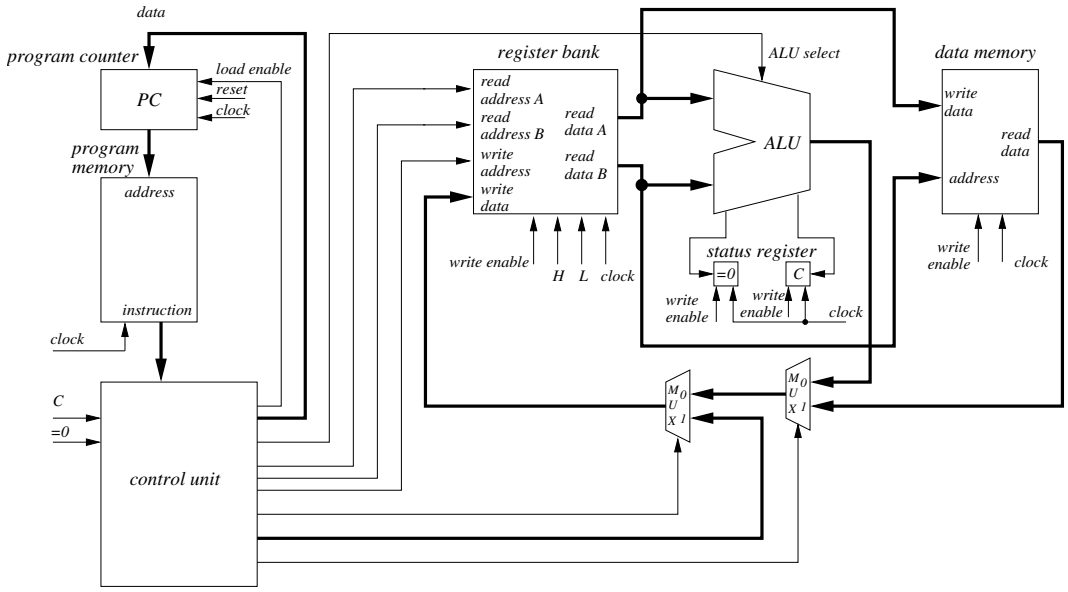

Figure 2. Structure of the RISC CPU

Table 1. Runtime of block-level verification

Block	Number of properties	Total run time in CPU seconds
register bank	4	1.03
program counter	3	0.08
control unit	11	0.23
data memory	2	0.49
program memory	2	0.48
ALU	17	4.41

2.3.1. Hardware

Properties for each block of the RISC CPU have been formulated. E.g. for the control unit it has been verified which control lines are set according to the opcode of the instruction input. Overall the correctness of each block could be verified. Table 1 summarizes the results[1].

The first column gives the name of the considered block. Next, the number of properties specified for a block are denoted. The last column provides the overall run time needed to prove all properties of a block. As can be seen the functional correctness of the hardware could be formally verified very fast with 39 properties.

2.3.2. Interface

Based on the hardware verification of the RISC CPU, in the next step the interface is verified. Thus, for each instruction of the RISC CPU a property has been specified which expresses the effects on all involved hardware blocks. As an example we discuss the verification of the ADD instruction.

Example 2. *Figure 3 gives details on the ADD instruction. Besides the assembler notation also the instruction format of the ADD instruction is shown. The specified property for the* ADD *instruction is shown in Figure 4. First of all the opcode and the three addresses of the registers are assigned to meaningful variables (lines 1-6). The assume part of the* ADD *property is defined from line 11 to 12 and states that there is no reset (line 11), the current instruction is addition (line 11) and the registers* $\mathrm{R}[0]$ *and* $\mathrm{R}[1]$ *are not addressed (since this register are special purpose registers). Under these assumptions we want to prove that in the next cycle the register* $\mathrm{R}[i]$ *(=reg.reg[**prev**(Ri_A)]) contains the sum of register* $\mathrm{R}[j]$ *and register* $\mathrm{R}[k]$ *(line 16), the carry (stat.C) in the status register is updated properly (line 16) and the zero bit (stat.Z) is set iff the result of the sum is zero (line 17). Furthermore we prove that the* ADD *instruction has no side effects, i.e. the contents of all registers which are different from* $\mathrm{R}[i]$ *are not modified.*

Analogously to the ADD instruction the complete instruction set of the RISC CPU has been verified. Table 2 summarizes the results. The first column gives the category of the instruction. In the second column the number of properties for each category is provided. The last column shows the total run time needed to prove all properties of a category. As can be seen the complete instruction set of the RISC CPU can be verified in less than 5 CPU minutes.

[1]For the verification in the synthesized model of the RISC CPU the sizes of the memories have been reduced.

```
1  OPCODE := instr[15:11];
2  Ri_A := instr[10:8];
3  Rj_A := instr[5:3];
4  Rk_A := instr[2:0];
5  Rj := reg.reg[Rj_A];
6  Rk := reg.reg[Rk_A];
7
8  property ADD
9  always
10  // assume part
11  ( reset = 0 && OPCODE = "00111" &&
12    Ri_A > 1 && Rj_A > 1 && Rk_A > 1 )
13  –>
14  // prove part
15   next(
16    (reg.reg[prev(Ri_A)] + (65536 * stat.C) = prev(Rj) + prev(Rk))
17     && ((reg.reg[prev(Ri_A)] = 0) <–> (stat.Z = 1))
18
19     // no side effects
20     && ((prev(Ri_A) != 2) –> reg.reg[2] = prev(reg.reg[2]) )
21     && ((prev(Ri_A) != 3) –> reg.reg[3] = prev(reg.reg[3]) )
22     && ((prev(Ri_A) != 4) –> reg.reg[4] = prev(reg.reg[4]) )
23     && ((prev(Ri_A) != 5) –> reg.reg[5] = prev(reg.reg[5]) )
24     && ((prev(Ri_A) != 6) –> reg.reg[6] = prev(reg.reg[6]) )
25     && ((prev(Ri_A) != 7) –> reg.reg[7] = prev(reg.reg[7]) )  );
```

Figure 4. Specified property for the ADD instruction of the RISC CPU

Assembler notation: ADD $R[i]$, $R[j]$, $R[k]$

Task: addition of $R[j]$ and $R[k]$, the result is stored in $R[i]$

Instruction format:

15 ... 11	10 9 8	7 6 5	4 3 2	1 0
0 0 1 1 1	*bin(i)*	- -	*bin(j)*	*bin(k)*

Figure 3. ADD instruction

Table 2. Runtime of interface verification

Instruction category	Number of properties	Total run time in CPU seconds
load/store instruction	6	15.16
arithmetic instructions	8	186.30
logic instructions	8	32.71
jump instructions	5	6.68
other instructions	5	7.14

2.3.3. Program

Finally, we describe the approach to verify assembler programs for the RISC CPU. As explained, the considered programs of the RISC CPU can be verified by constraining the instructions of the program as assumptions in the proof. These assumptions are automatically generated by the compiler of the RISC CPU. The verification of a program is illustrated in the following simple example:

```
1  /* counts from 10 downto 0  */
2          LDL R[7], 10
3          LDH R[7], 0
4  loop:
5          SUB R[7], R[7], R[1]
6          JNZ loop
```

Figure 5. Assembler program

Example 3. *Consider the assembler program shown in Figure 5. The program loads the integer* 10 *into register* R[7] *and decrements register* R[7] *in a loop until it contains value* 0*. For this program the property* count *has been formulated (see Figure 6). At first it is assumed that the CPU memory contains the instructions of the given example (lines 4 – 7)* [2]*. Furthermore the program counter points to the corresponding memory position (line 8), no memory*

[2] This part of the assumptions has been generated automatically by the compiler.

```
1  property count
2  always
3   // assume part
4   ( rom.mem[0] = 18186 &&  /* LDL  R[7], 10 */
5     rom.mem[1] = 20224 &&  /* LDH  R[7], 0 */
6     rom.mem[2] = 14137 &&  /* SUB  R[7], R[7], R[1] */
7     rom.mem[3] = 24578 &&  /* JNZ  2  */
8     pc.pc = 0 &&
9     next_a[0..21]( prog_mem_we = 0 ) &&
10    next_a[0..21]( reset = 0 ) )
11  ->
12  // prove part
13  next[21] (reg.reg[7] = 0 );
```

Figure 6. Property `count`

write operation is allowed (line 9) and there is no reset for the considered 22 *cycles (line 10). Under these assumptions we prove that the register* R[7] *is zero after* 21 *cycles. The time-point* 21 *results from the fact that the first two cycles (zero and one) are used by the load instructions and the following* 20 *cycles are required to loop* 10 *times. The complete proof has been carried out in less than 25 CPU seconds.*

3. Conclusions and Future Work

In this paper we presented an approach to hardware/ software co-verification based on bounded model checking. As a first example the method has been demonstrated for a RISC CPU. We succeeded to completely formally verify the hardware, the interface and simple programs.

Current studies on more difficult algorithms are very promising and it is focus of future work to consider more complex embedded systems based on our approach.

References

[1] *Accellera Property Specification Language Reference Manual, version 1.1*. http://www.pslsugar.org, 2005.

[2] B. Becker, R. Drechsler, and P. Molitor. *Technische Informatik — Eine Einführung*. Pearson Education Deutschland, 2005.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[4] P. Bjesse and K. Claessen. SAT-based verification without state space traversal. In *Formal Methods in Computer-Aided Design*, pages 372–389, 2000.

[5] D. Große and R. Drechsler. *CheckSyC*: An efficient property checker for RTL SystemC designs. In *IEEE International Symposium on Circuits and Systems*, pages 4167–4170, 2005.

[6] D. Große, U. Kühne, C. Genz, F. Schmiedle, B. Becker, R. Drechsler, and P. Molitor. Modellierung eines Mikroprozessors in SystemC. In *GI/ITG/GMM-Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 2005.

[7] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[8] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, 2000.

[9] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.

[10] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. A cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.