ICCD'93
International Conference on Computer Design
Cambridge, Massachusetts, USA, October 1993
IEEE Computer Society Press, pp. 58-62

Towards a Methodology for the Formal Hierarchical Verification of RISC Processors

Sofiène Tahar

University of Karlsruhe, Institute of Computer Design and Fault Tolerance (Prof. D. Schmid), P.O. Box 6980, 76128 Karlsruhe, Germany

Ramayya Kumar

Forschungszentrum Informatik, Department of Automation in Circuit Design, Haid-und-Neu Straße 10-14, 76131 Karlsruhe, Germany

Abstract

In this paper, a general methodology based on a hierarchical model of interpreters is presented for formally verifying RISC cores. The abstraction levels used by a designer in the implementation of RISC cores, namely the instruction set level, the pipeline stage level, the phase level and the hardware implementation, are mirrored by this hierarchical model. The use of this model allows us to successively prove the correctness between two neighbouring levels of abstractions, so that the verification process is simplified. The parallelism in the execution of the instructions, resulting from the pipelined architecture of RISCs is handled by splitting the proof into two simplified steps. The first step shows that, under certain assumptions, no conflicts can occur between simultaneously executed instructions and the second step shows that each instruction is implemented correctly by the sequential execution of its pipeline steps.

1. Introduction

In the recent past several successful microprocessor verification efforts have been performed; some using higher order logic [3,8,12] and others based on functional calculi [7,10]. These efforts handle very simplified processors, so-called toy machines, when compared with today's commercially available microprocessors.

Real CISC microprocessors have a very complex and unstructured design, including a large control part encoded in an intricate way. In contrast to CISCs, RISC designs are better structured and hence more tractable for performing formal verification. However, additional problems such as pipelining have to be tackled since they form the essence of RISCs. Furthermore, contemporary RISCs include complex features, such as floating point operations, memory management, etc. The architecture of such processors can be defined using a multiple layered architecture [2], consisting of the core, which can best be characterized as a RISC, and other layers, such as, the numeric architecture for floating point support and the protected architecture for memory management, multitasking and multiprocessing support. In this paper, we consider only the core architecture of a RISC processor.

The purpose of this work is the development of a generic methodology for the hierarchical verification of realistic RISC cores, that are not designed just for the purpose of verification. Furthermore, we are integrating this methodology into a general verification framework [9] and exercising it by means of a RISC example: DLX[1] [6].

The organization of this paper is as follows: Section 2 first sketches some related work and informally describes a novel hierarchical verification model for RISCs. Section 3 gives a formalization of the specification of this model. Section 4 explores ways of formally verifying this model in general and section 5 finally concludes the paper.

2. The hierarchical verification model

2.1 Related work

Most related work in microprocessor verification were concerned with microprogrammed processors [7,8,12]. Some recent work has shown that the verification of microprogrammed processors can be simplified by the insertion of intermediate abstraction levels, called *interpreters*, between the top-level specification given as an instruction set and the circuit implementation EBM (Electronic Block Model) [8,12]. These abstractions reflect the way complex microprocessor designs are carried out and designed [1]. Using this interpreter model, it is sufficient to prove that each level correctly implements the next abstraction level, instead of directly verifying that each architectural instruction is correctly implemented by the EBM. Through these appropriate intermediate levels, long and complex proofs are replaced by many more routine proofs, since the gap between the neighbouring levels is small.

The way RISC designs are carried out and structured is different from that of CISCs, e.g. because of the hardwired

1. DLX is an hypothetical RISC which includes the most common features of some existing RISCs (Intel i860, Motorola M88000, SPARC, MIPS R3000).

control the micro level does not exist more. The mentioned structuring of the verification using this interpreter model is hence unsuitable and we have to look for another verification model for RISCs.

2.2 The RISC verification model

A RISC processor executes every architectural instruction in a number of physical steps, called pipeline stages (e.g. IF, ID, EX, WB) [6]. The duration of a pipeline stage instruction corresponds to one machine clock period. A *stage instruction* is the set of transfers, which occur during the corresponding pipeline segment. Using a multiple phase non-overlapping clock, each stage instruction is partitioned into a number of clock phase instructions. A *phase instruction* of a specific stage is a part of the stage instruction that occurs during that clock phase. However, in contrast to microprogrammed processors, the RISC phase instructions are stage dependent and the stage instructions differ from one instruction to another. This implies that the number of phase instructions and consequently the verification tasks between the EBM and the phase level are big.

As a first solution to reduce this number, we exploit the notion of instruction classes. An *instruction class* intuitively corresponds to the set of instructions with similar semantics, e.g. ALU, FLP, LOAD, CONTROL. The stage and phase instructions can now be parameterized in accordance to the class abstraction, i.e. they are not dependent on each instruction but only on the instruction class. The class level is therefore introduced as the top level of our interpreter model.

	ALU	LOAD	STORE	CONTROL
IF	$IR \leftarrow M[PC]$ $PC \leftarrow PC+4$	$IR \leftarrow M[PC]$ $PC \leftarrow PC+4$	$IR \leftarrow M[PC]$ $PC \leftarrow PC+4$	$IR \leftarrow M[PC]$ $PC \leftarrow PC+4$
ID	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ $IR1 \leftarrow IR$	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ $IR1 \leftarrow IR$	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ $IR1 \leftarrow IR$	$BTA \xleftarrow[\phi_1]{} f(PC)$ $PC \xleftarrow[\phi_2]{} BTA$
EX	$Aluout \leftarrow A \text{ op } B$	$MAR \leftarrow A+(IR1)$	$MAR \leftarrow A+(IR1)$ $SMDR \leftarrow B$	
MEM	$Aluout \leftarrow Aluout1$	$LMDR \leftarrow M[MAR]$	$M[MAR] \leftarrow SMDR$	
WB	$RF[rd] \xleftarrow[\phi_1]{} Aluout1$	$RF[rd] \xleftarrow[\phi_1]{} LMDR$		

Table 1: DLX Pipeline Structure

Real RISCs show further regularities which can be incorporated into our interpreter model. A closer look at the stage level (see table 1) shows that some stage instructions are commonly used by more than one class (e.g. the IF-stage instruction is shared by all classes), and additionally some classes do not require the full pipeline depth (e.g. only 2 stages for CONTROL). Furthermore, examining the phase level of realistic RISCs, it can be seen that not all stage operations are broken down into phases. Such phase level instructions can be modelled by letting the state of the interpreter at the phase level unchanged. Incorporating all these observations into our model yields a reduced number of different possible stage and phase instructions. Summarizing the overall specification of the different abstraction levels, we have to specify:

- the class level from the instruction set of the RISC,
- the stage level from the pipeline architecture,
- the phase level from the pipeline architecture and
- a formal implementation of EBM.

Considering only a sequential execution of the instructions within our interpreter model, the overall verification process is partitioned hierarchically as shown in figure 1, where the arrow between the levels means that the lower level specifications imply all instructions of the next upper one.

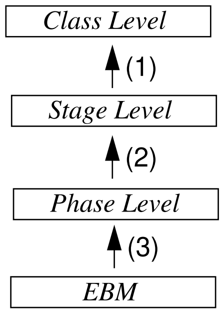

Figure 1: RISC Interpreter Model

Using the steps (1),(2) and (3), we are able to successively prove the correctness of the class, stage and phase levels. Through this hierarchical structuring of the verification steps, we have closed the big gap between the EBM and the architecture top level. Step (3) is the hardest one, since the EBM is a complex structural description and the phase instructions are behavioural ones. However, the proofs are quite similar in nature and a strategy can therefore be evolved. Steps (1) and (2) are relatively straightforward as illustrated in section 4. Each verification step can be done independently and should help the designer in successively refining and verifying the design.

Having shown the correctness of the class level, the architectural instructions can then be proven correct by a simple *instantiation* of the previous steps (1, 2 and 3) for each particular instruction.

3. Formal specification

Each abstraction level, also called an interpreter, consists of the set of visible states and a set of state transitions describing the semantics of the instructions corresponding to that abstract level. In moving from the class level abstraction to the EBM, refinements occur in the set of visible states and also in the temporal domain. The temporal unit at the different levels are the instruction cycle time, clock cycle time and the phase time of the clock [11].

The instruction semantics, at each level of abstraction, can be formally defined as a predicate whose parameters correspond to the visible states at that level of abstraction. In the description to follow, the notation "*x:ty*" means that "*x*" is of type "*ty*")

3.1 Class level

The instruction semantics given in the instruction set manual of a specific processor implicitly involve time. For example the semantic of an ADD-instruction, defined as a state transition occurring in an instruction cycle is:

$$ADD := RF[rd] \leftarrow RF[rs1] + RF[rs2]$$

The class level is only an abstraction of the semantic of the architectural instructions. Hence, the semantic of the ALU-class can be given as follows:

$$ALU := RF[rd] \leftarrow RF[rs1] \; \mathbf{op} \; RF[rs2]$$

Given that u is a unit for an instruction cycle, the specification of the ALU-instruction involving time can be described formally by the following predicate, where *PC, I-MEM, RF* and *D-MEM* are the program counter, the instruction memory, the register file and the data memory, respectively:

$$\begin{aligned}
&ALU_SPEC\ (PC,\ I\text{-}MEM,\ RF,\ D\text{-}MEM,\ \mathbf{op}) := \\
&\quad \forall\, u{:}Inst_cycle. \\
&\quad\quad RF(u+1)[rd(u)] = RF(u)[rs1(u)] \; \mathbf{op} \; RF(u)[rs2(u)]
\end{aligned}$$

3.2 Stage level

The stage predicates are built up of conjunctions of elementary state transitions, that implement the corresponding semantic. The additional visible states at the stage level are mostly pipeline buffer latches as shown in table 1. The state transitions can be directly read from the pipeline architecture (table 1) and encoded formally to build the specification predicate of the corresponding stage instruction, e.g. from the ID-row the common ID-stage instruction can be directly specified by the following predicate:

$$\begin{aligned}
&ID_A_SPEC\ (A,\ B,\ RF,\ IR,\ IR1) := \\
&\quad \forall t{:}Clk_cycle.\ \ A(t+1) = RF(t)\ [rs1(t)] \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\ B(t+1) = RF(t)\ [rs2(t)] \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\ IR1(t+1) = IR(t)
\end{aligned}$$

3.3 Phase level

Similarly to the stage instructions, the phase predicates are built up of conjunctions of elementary state transitions and these can be directly read from the pipeline architecture (table 1) and encoded formally. For phase transitions that are not explicitly marked in the pipeline architecture, e.g. between IR and IR1-register in the ID-stage, we simply let the state values remain unchanged until the last phase. As a refinement of the ID-stage specification given above, the two corresponding ID-phase instructions can be directly specified form the ID-row by the following predicates:

$$\begin{aligned}
&\phi_1 ID_A_SPEC\ (A,\ B,\ RF,\ IR,\ IR1) := \\
&\quad \forall \tau{:}Clk_phase.\ \ IR(\tau+1) = IR(\tau)
\end{aligned}$$

$$\begin{aligned}
&\phi_2 ID_A_SPEC\ (A,\ B,\ RF,\ IR,\ IR1) := \\
&\quad \forall \tau{:}Clk_phase.\ \ A(\tau+1) = RF(\tau)[rs1(\tau)] \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\ B(\tau+1) = RF(\tau)[rs2(\tau)] \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\ IR1(\tau+1) = IR(\tau)
\end{aligned}$$

3.4 Electronic Block Model

This level of specification is different from the others in that it specifies the structure of the circuit and not the behaviour. The EBM is in general structured hierarchically at the RT-level. At the top most level of this hierarchy, the EBM is specified formally by a conjunction of predicates for the microprocessor, which is built up of a datapath and a control unit, the instruction and data memory, respectively. The datapath and the control unit are themselves compositions of simpler blocks, e.g. register file, ALU, multiplexers, pipeline latches, etc., which may again be conjunctions of lower building blocks. The formal implementation of the EBM at the top most level, a la Hanna and Daeche [5] is as follows:

$$\begin{aligned}
&EBM\ (PC,\ I\text{-}MEM,\ RF,\ldots,\ A,\ B,\ Aluout,\ldots,\ IR,\ IR1,\ \ldots) = \\
&\quad \exists\ rs1, rs2, rd, alu_op, rw,\ imem_adr, \ldots\ . \\
&\quad\quad DataPath\ (RF,\ A,\ B,\ Aluout,\ alu_op,\ \ldots) \wedge \\
&\quad\quad Control_Unit\ (PC,\ IR,\ rw,\ imem_adr,\ \ldots) \wedge \\
&\quad\quad Instr_Memory\ (I\text{-}MEM,\ imem_adr,\ \ldots) \wedge \\
&\quad\quad Data_Memory\ (D\text{-}MEM,\ dmem_adr,\ rw,\ \ldots)
\end{aligned}$$

4. Formal verification

The aim of the formal verification is to show that the instruction set is executed correctly by the EBM, in spite of the pipelined architecture. At any clock cycle the processor can potentially be executing n_s instructions in parallel, in n_s different stages (where n_s is the pipeline depth). Data/control dependencies could exist between these instructions leading to interferences. Thus the correctness of the instruction semantics should necessarily incorporate the temporal aspects of the execution.

A possible approach for verification is to perform the correctness proof in two steps as follows:

1. given the implementation EBM, which includes hardware for some conflict resolutions, and assuming some software conditions of the actual architecture (represented by *SW_Ass*), we can prove that none of the possible conflicts occur, i.e. the predicate *Conflict_Pred* is never true:

   $$SW\_Ass,\ EBM\ \vdash\ \neg Conflict\_Pred$$

2. assuming that no conflicts occur during the execution of any instruction, the EBM implements each single instruction (instruction level) correctly:

   $$\neg Conflict\_Pred\ \vdash\ EBM \Rightarrow Instruction\ Level$$

These correctness proofs are elaborated in the subsections to follow.

4.1 Conflict freedom proof

In this first part of the verification process, we will deal with the proof, that at all times the parallel execution of instructions does not lead to any conflicts. There are three classes of conflicts (also called hazards), that can appear during the pipelined execution of any RISC machine [6]: structural hazards, data hazards and control hazards. All these hazards have to be formally specified in form of predicates and should be proven false.

Structural hazards correspond to resource conflicts and are defined as a conjunction of two predicates at the stage and phase level, respectively. The existence of a structural hazard at the stage level can be defined as follows:

$$\begin{aligned}
&Stage_Struct_Pred :=\\
&\quad \exists r{:}\ Resource.\ \exists S_i, S_j{:}\ Stage_Instruction.\\
&\qquad (\ S_type(S_i) \neq S_type(S_j)\ ) \wedge\\
&\qquad S_used_Pred(S_i,r) \wedge S_used_Pred(S_j,r)
\end{aligned}$$

where $S\_type(S_i)$ is a function which gives the logical type of the stage instruction S_i, e.g. ID or EX, and $S\_used\_Pred(S_i,r)$ means that the resource r, visible at the stage level, is used by the instruction S_i. The structural hazard at the phase level can be defined in a similar manner.

Data hazards are of three types called, read after write (RAW), write after read (WAR) and write after write (WAW). They all indicate the usage of a common storage element between two data dependent instructions [6]. A possible definition of the RAW-predicate is:

$$\begin{aligned}
&RAW_Data_Pred :=\\
&\quad \exists r{:}\ Resource.\ \exists C_i, C_j{:}\ Class_Instruction.\\
&\quad \exists p_i, p_j{:}\ Stage.\ \exists t_i, t_j{:}\ Clk_cycle.\\
&\qquad (0 < (j\text{-}i) < n_s) \wedge (t_i \geq t_j) \wedge\\
&\qquad Range(r, C_i, p_i, t_i) \wedge Domain(r, C_j, p_j, t_j)
\end{aligned}$$

where $Range(r, C_i, p_i, t_i)$ implies that the resource r, visible at the class level, is written by the instruction C_i in pipeline stage p_i at time t_i and $Domain(r, C_j, p_j, t_j)$ implies that the resource r is read by the instruction C_j in pipeline stage p_j at time t_j. In a similar manner the WAR and WAW predicates can be defined.

Control hazards involve all kinds of interrupts to the linear pipeline flow. This includes software Branches, Jumps, Traps instructions, as hardware interrupts, stalls, etc. The behaviour of each of these cases have to be also formally specified in form of predicates.

Having specified the various hazard predicates, assuming some hardware and software constraints for implementing conflict resolution mechanisms (e.g. the number of delay slots when using a software scheduling technique), and given the specification of the interpreter levels and EBM, the verification step comprises the proof that none of the conflict predicates is ever true.

4.2 Correctness of single instructions

Using the notion of instruction classes and the hierarchical interpreter model, as described briefly in section 2, we first successively prove the correctness of all class instructions, i.e.:

$$EBM \overset{3}{\Rightarrow} Phase\ Level \overset{2}{\Rightarrow} Stage\ Level \overset{1}{\Rightarrow} Class\ Level$$

and then the obtained proof theorems are instantiated for each instruction of the architecture. The verification steps (1, 2 and 3) can be done independently (as mentioned in section 2.2) and are described in the following subsections.

Class level: In order to show the correctness of the class level, we have to prove that individual class specifications are correctly implemented by the sequential execution of their corresponding stage instructions. As seen in section 3, these abstraction levels include different temporal abstractions. These have to be related to each other using a temporal abstraction function. In a pipelined execution this mapping function is not linear and depends on the pipeline architecture. Therefore, the temporal abstraction function also depends on a context parameter, which includes read/write information and an indication of the corresponding pipeline stage [11]. Hence the abstract specification is modified such that existentially quantified context parameters are introduced and the time abstraction function (represented below by *f*) is applied on the abstract time variable. The extended specification of the ALU-class instruction (see section 3.1) looks formally as follows:

$$\begin{aligned}
&ALU_SPEC^*\ (PC, I\text{-}MEM, RF, D\text{-}MEM, \mathbf{op}) :=\\
&\quad \forall u{:}Inst_cycle.\ \ \exists c1, c2, c3{:}Context.\\
&\qquad RF(f(c1,u))\ [rd(f(c2,u))] = RF(f(c3,u))\ [rs1(f(c2,u))]\\
&\qquad\qquad \mathbf{op}\ RF(f(c3,u))\ [rs2(f(c2,u))]
\end{aligned}$$

The context parameters are directly derived from the implemented pipeline architecture (table 1), e.g. since the register file is *read* at the *ID-stage* the time abstraction function is applied to it using the context parameter *c3 = [R;ID]*. The verification goal for the ALU-class instruction is hence given as follows:

$$
\begin{aligned}
\vdash\ \forall op.\ \ & \mathit{IF\_SPEC}\ (PC,\ I\text{-}MEM,\ IR) \wedge \\
& \mathit{ID\_SPEC}\ (A,\ B,\ RF,\ IR,\ IR1) \wedge \\
& \mathit{EX\_SPEC}\ (A,\ B,\ Aluout,\ IR1, IR2,\ \mathbf{op}) \wedge \\
& \mathit{MEM\_SPEC}\ (Aluout1,\ Aluout,\ IR2,\ IR3) \wedge \\
& \mathit{WB\_SPEC}\ (RF,\ Aluout1,\ IR3) \\
\Rightarrow\ & \mathit{ALU\_SPEC^{*}}\ (PC,\ I\text{-}MEM,\ RF,\ D\text{-}MEM,\ \mathbf{op})
\end{aligned}
$$

Setting the goal and deriving the proof of the resulting theorem are done automatically through a generic function, that takes the corresponding specification predicates, context informations, the pipeline depth and the stage identifiers as parameters.

Stage level: Since the verification goals for the correctness of the stage level are similar (the conjunction of phase instructions implies a stage instruction), the same temporal abstraction mechanism and the same generic function for goal settings and proofs are used. However, the appropriate parameters, e.g. number of clock phases, phase identifiers, etc., have to be set accordingly.

Phase level: Due to the advantage of having used the hierarchical interpreter model, we only have to show the correctness of a small number of phase instructions, build up of simple transitions as seen in section 3.3. Although the EBM implementation is very large and quite complex, a great amount of automation has been achieved in this domain via MEPHISTO [9]. The goal to be proved is split into appropriate subgoals which can be solved more or less automatically by the MEPHISTO verification framework. For the correctness of the phase level we use a generic function which generates the verification goals and does the proofs automatically, using an appropriate general proof schema based on MEPHISTO.

5. Conclusions, results and future work

In this paper we have described a hierarchical interpreter model which is applicable for RISC cores in general. This model reflects the design hierarchy which is used for designing RISCs. Hence, the methodology based on it can be used by computer architecture designers for formally verifying their designs during the design phase itself. The correctness of the pipelined execution of each instruction is also ensured by splitting the proof goal into two parts, namely conflict freedom during the simultaneous execution of various instructions and the correct implementation of each instruction by its pipelined stages. The methodology is generic, in that it provides a pattern to follow when verifying RISC processors. The specification and verification steps were kept general and implementation-independent.

We have also implemented the hardware for the DLX core in a commercial VLSI design environment (using a 1.5 μm CMOS technology, the design has a size of approximately 100.000 transistors and runs at a clock rate of 16 MHz). It is being used as one of the benchmarks for our methodology.

We have implemented different specifications (section 3) and proof strategies (section 4.2) in HOL [4] (the overall specification text is 492 lines long, the goal settings and proofs for the whole DLX processor core took 550.048 s on a SPARC10 with a 96 MB main memory). Proof strategies for the multiple conflict cases are currently under development.

References

[1] Anceau, F.: The Architecture of Microprocessors; Addison-Wesley Publishing Company, 1986.

[2] Baker, T.: Headroom and Legroom in the 80960 Architecture; COMPCON90, 35th. IEEE Computer Society International Conference, San Francisco, Feb.-March, 1990.

[3] Cohn, A.: A Proof of the Viper Microprocessor: The First Level; In: VLSI Specification, Verification and Synthesis, Eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1988.

[4] Gordon, M.: A Proof Generating System for Higher-Order Logic; In: VLSI Specification, Verification and Synthesis, Eds. G. Birtwistle and P. A. Subrahmanyam, Kluwer, 1988.

[5] Hanna, F., Daeche, N.: Specification and Verification of Digital Systems Using Higher-Order Predicate Logic; IEE Proc. Pt. E, Vol. 133, No. 3, September 1986, pp. 242-254.

[6] Hennessy, J., Patterson, D.: Computer Architecture A Quantitative Approach; Morgan Kaufmann Publishers, Inc. San Mateo, California, 1990.

[7] Hunt, W.: The Mechanical Verification of a Microprocessor Design; In: From HDL Description to Guaranteed Correct Circuit Designs, Ed. D. Borrione, North-Holland, 1987.

[8] Joyce, J.: Multi-Level Verification of Microprocessor-Based Systems; PhD thesis, Cambridge University, December 1989.

[9] Kumar, R., Schneider, K., Kropf, Th.: Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment; Journal of Formal Methods in System Design, Vol. 2, pp. 165-230, 1993.

[10] Srivas, M., Bickford, M.: Verification of a Pipelined Microprocessor Using Clio; In: Hardware Specification, Verification and Synthesis: Mathematical Aspects, Eds. M. Leeser and G. Brown, Springer, 1990.

[11] Tahar, S., Kumar, R.: A Formalization of a Hierarchical Model for RISC Processors; to appear in Proc. of Euro-ARCH'93, Munich, Germany, Springer Verlag, 1993.

[12] Windley, P.: The Formal Verification of Generic Interpreters; PhD thesis, University of California, Davis, Division of Computer Science, July 1990. Research Report CSE-90-22.