The Computer Journal
Vol. 38, No. 2, 1995
Oxford University Press, pp. 111-120

Formal Specification and Verification Techniques for RISC Pipeline Conflicts

Sofiène Tahar and Ramayya Kumar*
Institut für Rechnerentwurf und Fehlertoleranz, Universität Karlsruhe, Germany
* Forschungszentrum Informatik, Karlsruhe, Germany

We outline a general methodology for the formal verification of instruction pipelines in RISC cores. The different kinds of conflicts, i. e. resource, data and control conflicts that can occur due to the simultaneous execution of the instructions in the pipeline, have been formally specified in higher order logic. Based on a hierarchical model for RISC processors, we have developed a constructive proof methodology, i.e. when conflicts at a specific abstraction level are detected, the conditions under which these occur are generated and explicitly output to the designer, thus easing their removal. All implemented specifications and tactics are kept general, so that they are usable for a wide range of RISC cores. In this paper, the described formalization and proof strategies are illustrated via the DLX RISC processor.

1 Introduction

Formal verification of processors have gained considerable attention in the recent past. However, most of the work has concentrated on the verification of microprogrammed processors [3, 9, 10, 20]. The objective of our endeavour is the development of a generic methodology for the hierarchical verification of a large number of realistic RISC processors cores. The previous work in the verification of RISC processors were only able to verify parts of processors at certain levels of abstraction [2, 14]. In contrast, we are developing a methodology and an associated environment for the routine verification of RISC cores in its entirety, i.e. from the specification of instruction sets down to their circuit implementations. In our previous work, we have constructed a hierarchical model comprising of various abstraction levels, which closely corresponds to the hierarchy used in the design of RISC cores [16]. Using this model, higher-order logic specifications at various abstraction levels can be given in a straightforward manner. These formal specifications are then used in conjunction with parameterized proof scripts which automate the verification process [17]. Parts of the formal proofs which do not deal with the conflicts occurring due to pipelining, have been described in [18] and the concepts of pipeline verification have been described in [19].

In this paper, we focus our attention on the formalization and proofs pertaining to pipeline conflicts. These conflicts occur due to the data and control dependencies and resource contentions caused by the simultaneous executions of the instructions in the pipeline. We describe automatic, constructive proof techniques for conflict detections where the conditions under which these conflicts occur are explicitly stated. This aids the designer in easily formulating the conflict resolution mechanisms, either in hardware or software.

The organization of this paper is as follows. Section 2 briefly presents the hierarchical model on which the formal verification methodology is based. Section 3 gives an overview of the overall verification process. Section 4 includes some formal definitions of types and functions used in the formal specifications that follow. Sections 5 through 7 define the conflicts arising due to the pipelined nature of RISCs and describe the correctness proofs for resource, data and control conflicts, respectively. Section 8 contains some experimental results and section 9 concludes the paper. All the methods and techniques presented in this paper are illustrated by means of a RISC processor called, DLX[1] [8].

2 RISC Verification Model

Our RISC verification model is closely related to the interpreter model of Anceau [1] for the design and description of microprogrammed processors. This model has been adapted for the formal verification of microprogrammed processors by Windley [20]. Instead of directly showing that the implemented circuit (Electronic Block Model – EBM) correctly implements each instruction, he has shown the correctness between the neighbouring abstraction levels and thus reduced the complexity of the verification process. Although the RISC processors also possess similar levels of hierarchy in the design, a naive mapping of this interpreter model onto RISCs does not reduce the complexity of the verification process. This is due to the fact that, in contrast to microprogrammed processors, the temporal abstractions between the hierarchy levels are non-linear and additionally, the instructions at different abstraction levels in RISCs are dependent on each other, as shown in [16].

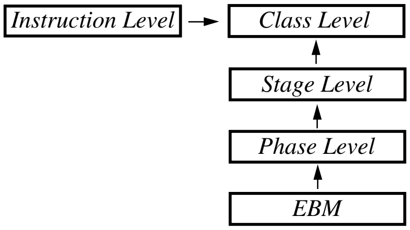

Figure 1: RISC Interpreter Model

The RISC interpreter model comprises the instruction, class, stage and phase levels, each of which corresponds to an interpreter at different abstraction levels, and the lowest level EBM, which corresponds to the circuit implementation (figure 1). Each interpreter consists of a set of visible states and a state transition function which defines the semantics of the interpreter at that level of abstraction. Between two levels, a structural abstraction (set of visible states), a

---

[1] DLX is an hypothetical RISC which includes the most common features of existing RISCs such as Intel i860, Motorola M88000, Sun SPARC or MIPS R3000.

behavioural abstraction (functional semantics), a temporal abstraction (level of time granularity) and a data abstraction (level of data granularity) may exist. At the instruction level, states such as the program counter, register file, instruction and data memories, etc. are visible and the set of transition functions corresponds to the instruction set of the RISC core. The class level is essentially an abstraction of the different instructions into instruction classes such as, the arithmetic instructions, store instructions, load instructions, control instructions, etc. The set of visible states are the same as that of the instruction level and the state transfers are abstractions of the instruction set. For example, all binary arithmetic and logic operations at the instruction level are replaced by a single operation called "op" (row EX and column ALU as shown in table 1). The temporal granularity at these two levels is that of an instruction cycle. At the stage and phase levels more states, corresponding to the refined implementations are seen and the time granularities are that of a clock cycle and a clock phase duration, respectively. A stage (phase) instruction is defined as the set of elementary transfers of one or more class instructions at a specific pipeline stage (clock phase), as illustrated in table 1 for the DLX example. The EBM corresponds to the circuit implementation at the register-transfer level.

Table 1: .DLX Pipeline Structure

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

3 Verification Tasks

Starting from the instruction set of a RISC processor, we have to show that this instruction set is executed correctly by the EBM, in spite of the pipelined architecture. A RISC processor executes n_s instructions in parallel (see figure 2), in n_s different pipeline stages. This parallel execution increases the overall throughput of the processor; however, it should be noted that no single instruction runs faster. Due to this parallel execution of a certain number of instruction, we should rather verify the dynamic behaviour of a stream of instructions instead of just considering the behaviour of the instruction set. Hence, we have to prove, on the one hand, that the sequential execution of each instruction is correctly implemented by the EBM and on the other hand, to show that the pipelined execution of the instructions is correct. Thus the correctness proof is split into two independent steps as follows:

1. given some software constraints on the actual architecture and given the implementation EBM, we prove that any sequence of instructions is correctly pipelined, i.e.:

$$\text{SW\_Constraints, EBM} \vdash \text{Correct\_Instr\_Pipelining} \qquad \textbf{(1)}$$

2. we prove that the EBM implements the semantics of each single architectural instruction correctly, i.e.:

$$\vdash \text{EBM} \Rightarrow \textit{Instruction Level} \qquad \textbf{(2)}$$

The software constraints in (1) are part of the top level specification of the RISC processor and represent those conditions which are to be met for designing the software so as to avoid conflicts, e.g. the number of delay slots to be introduced between the instructions while using a software scheduling technique. Additionally, it is also assumed that the EBM includes some conflict resolution mechanisms in hardware.

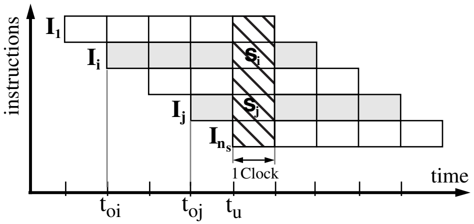

Figure 2: Pipelined Execution

According to the nature of the verification step to be handled, we call step (1) the verification of pipeline correctness and step (2) the verification of the semantic correctness. The proof of semantic correctness has been discussed and reported in our previous work [17, 18] and we recapitulate it briefly in the next subsection. The pipeline correctness is the main topic of this paper and will be discussed in detail, in the rest of the paper.

3.1 Semantic Correctness

In order to verify the semantic correctness, the higher-order logic specifications and implementations at the various levels of abstraction are used and the following is proved — *EBM ⇒ Phase Level ⇒ Stage Level ⇒ Class Level*. This proof is similar to the one done by Windley in proving the correctness of microprogrammed processors [20]. Each obtained theorem is then instantiated for each instruction at the instruction level. We have implemented tactics in the HOL theorem prover [6], using the hardware verification environment — MEPHISTO [12], which automates the entire process, given the formal definitions of the specifications and implementations at each abstraction level [16].

3.2 Pipeline Correctness

The pipeline correctness can be proved by showing that all possible combinations of n_s instructions are executed correctly. This implies that the parallel execution of n_s instructions does not lead to any *conflicts*. There are three kinds of conflicts (also called *hazards*) that can occur during the pipelined execution of any RISC machine namely, resource, data and control conflicts[2]. The predicate Correct_Instr_Pipelining in (1), which implies the absence of these conflicts, is defined as the conjunction of predicates for each conflict kind. Formally:

$$\boxed{\vdash_{def}\ \text{Correct\_Instr\_Pipelining} := \begin{array}{l} \neg\,\text{Res\_Conflict} \wedge \\ \neg\,\text{Data\_Conflict} \wedge \\ \neg\,\text{Control\_Conflict} \end{array}}$$

and the correctness statement (1) can be rewritten as:

$$\boxed{\text{SW\_Constraints, EBM} \ \vdash \begin{array}{l} (\neg\,\text{Res\_Conflict} \wedge \\ \ \neg\,\text{Data\_Conflict} \wedge \\ \ \neg\,\text{Control\_Conflict}) \end{array}}$$

thus the whole correctness proof is tackled by splitting it into three independent parts, each corresponding to one kind of conflict. These parts are elaborated in the sections to follow.

In proving the pipeline correctness, we have to ensure that all possible combinations of instructions occurring in n_s stages are executed correctly. This large number can be reduced by an order of magnitude when the notion of classes is exploited. When conflicts are formalized at the class, stage and phase levels, the existence of multiple instructions in the pipeline can be formalized by predicates which we call the *multiple conflict predicates*. Their proofs are further simplified by constructing conflict predicates between pairs of instructions which are called the *dual conflict predicates*. These notions will be clarified in the sections to follow.

4 Formal Definitions

In our previous papers, we have given a detailed formalization of instructions at all abstraction levels [16, 18]. In this section, we briefly introduce some processor specific types and generalized functions that are useful for handling conflict predicates. The entire process of extracting the formal specifications can be completely automated, except for certain processor specific type definitions. According to our hierarchical model, we define for each abstraction level a processor specific set of enumeration types for corresponding instructions, resources and pipeline characteristics, i.e. pipeline stages or clock phases. Referring to the pipeline structure of table 1, where the columns represent the four instruction classes — ALU, LOAD, STORE and CONTROL and the rows correspond to the related stage and phase transfers at the five pipeline stages — IF, ID, EX, MEM and WB and the two clock phases ϕ_1 and ϕ_2, respectively, we define the following enumeration types for the DLX example:

- types for pipeline stages and clock phases:

  `type` "pipeline_stage = IF | ID | EX | MEM | WB "
  `type` "clock_phase = Ph1 | Ph2"

- types for the set of all instructions at each abstraction level:

  `type` "class_instruction = ALU | LOAD | STORE | CONTROL"
  `type` "stage_instruction = IF_X | ID_X | ID_C | EX_A | EX_I | … | WB_L"
  `type` "phase_instruction = Ph1_IF_X | … | Ph1_EX_A | … | Ph2_WB_L"

- types for resources (related to the structural abstraction, where CL, SL and PL stand for Class Level, Stage Level and Phase Level, respectively):

  `type` "CL_resource = PC | RF of RF_addr | I_MEM | D_MEM | IAR"
  `type` "SL_resource = PC | RF of RF_addr | I_MEM | … | IR | A | B | Aluout | …"
  `type` "PL_resource = PC | RF of RF_addr | … | IR | A | B | Aluout | … | BTA"

For the specification of the conflict predicates, we have defined several generalized functions and predicates as follows:

- abstraction functions, which either compute higher level instructions from lower ones or extract lower level instructions from higher ones:

  ClassToStage : ((pipeline_stage, class_instruction) → stage_instruction)
  StageToClass : (stage_instruction → class_instruction)
  StageToPhase : ((clock_phase, stage_instruction) → phase_instruction)
  PhaseToStage : (phase_instruction → stage_instruction)

  e.g. ClassToStage (ID, CONTROL) = ID_C,
  PhaseToStage (Ph2_ID_C) = ID_C.

- functions that compute the logical pipeline stage or clock phase types from a stage or a phase instruction, respectively:

  Stage_Type : (stage_instruction → pipeline_stage)
  Phase_Type : (phase_instruction → clock_phase)

- functions which compute the ordinal values of a given pipeline stage and clock phase, respectively[3]:

  Stage_Rank : (pipeline_stage → num)
  Phase_Rank : (clock_phase → num)
  e.g. Stage_Rank (ID) = 2, Phase_Rank (Ph1) = 1.

- predicates, which are true if a given resource is used by a given stage and phase instruction, respectively:

  Stage_Used : ((stage_instruction, SL_resource) → bool)
  Phase_Used : ((phase_instruction, CL_resource) → bool)

  e.g. Stage_Used (ID_C, PC) = *True*. These Predicates are automatically extracted from the formal specifications of the stage and phase levels, respectively (refer to [18]).

- predicates that imply that a given resource is read (*domain*) or written (*range*) by a given class or stage instruction at a given pipeline stage or clock phase, respectively:

  Stage_Domain : ((class_instruction, pipeline_stage, CL_resource) → bool)
  Stage_Range : ((class_instruction, pipeline_stage, CL_resource) → bool)
  Phase_Domain : ((stage_instruction, clock_phase, CL_resource) → bool)
  Phase_Range : ((stage_instruction, clock_phase, CL_resource) → bool)

---

[2] A formal definition of these conflicts is given in later sections.

[3] These functions are needed to express the sequential order of the execution of stage and phase instructions.

e.g. Stage_Domain (ALU, ID, RF) = *True*, Phase_Range (ID_C, Ph2, D_MEM) = *False* (refer also to table 1). These predicates are automatically extracted from the specification of the class and stage level instructions at the clock cycle and clock phase time granularities, respectively (refer to [18]).

Each of the above predicates (Stage/Phase_Used, Stage/Phase_Domain and Stage/Phase_Range) is generated as a theorem for the given combination of class, stage or phase instruction, pipeline stage or clock phase and level corresponding resource, respectively. All these theorems are created once and put in appropriate lists which will be used for rewriting later during the verification of resource data and control conflicts.

5 Resource Conflicts

Resource conflicts (also called *structural hazards* [8, 11, 13, 15] or *collisions* [11, 15]) arise when the hardware cannot support all possible combinations of instructions during the simultaneous overlapped execution. This occurs when some resources or functional units are not duplicated enough and two or more instructions attempt to use them simultaneously. A resource could be a register, a memory unit, a functional unit, a bus, etc. The use of a resource is a write operation for storage elements and an allocation for functional units. In the sections to follow, we will first formally specify the resource conflicts and then discuss the correctness proof issues.

5.1 Resource Conflict Specification

Referring to the hierarchical model (see figure 1), the formal specifications of resource conflicts are handled according to the different abstract levels. Furthermore, only the visible resources related to each abstraction level are considered by the corresponding resource conflict predicates.

Class Level Conflicts. The resource conflict predicate Res_Conflict, as mentioned in section 3.2, is equivalent to a multiple conflict between the maximal number of class instructions that occur in the pipeline. This Multiple_Res_Conflict predicate is true if any pair of the corresponding stage instructions compete for one resource (see hatched box in figure 2). Formally, Multiple_Res_Conflict is defined in terms of disjunctions over all possible stage instruction pair conflicts, which correspond to the class instructions $I_1 \ldots I_{n_s}$. Let Dual_Stage_Conflict be a predicate describing dual stage conflicts, the multiple resource conflict is specified formally in terms of dual conflicts as follows (where depending on the index i, ψ_i represents the related pipeline stage, i.e. ψ_1 = IF, ψ_2 = ID, ψ_3 = EX, etc.):

$$\begin{aligned}
&\text{Res\_Conflict} = \\
&\quad \vdash_{def}\ \text{Multiple\_Res\_Conflict}\ (I_1, \ldots, I_{n_s}) := \\
&\quad \bigvee_{\substack{i,j \\ (i,\, j = 1 \ldots n_s) \\ (i < j)}} \text{Dual\_Stage\_Conflict}\ (\text{ClassToStage}\ (\psi_i, I_{n_s - i + 1}), \\
&\qquad\qquad\qquad\qquad\qquad \text{ClassToStage}\ (\psi_j, I_{n_s - j + 1}))
\end{aligned}$$

Stage Level Conflicts. A dual resource conflict happens when two stage instructions attempt to use the same resource. Furthermore, since only stage instructions of different types can be executed simultaneously in the pipeline (see hatched box in figure 2), we should ensure that the corresponding stages are of different logical types. Formally, the Dual_Stage_Conflict predicate is specified using the function Stage_Type and the predicate Stage_Used as follows:

$$\begin{aligned}
&\vdash_{def}\ \text{Dual\_Stage\_Conflict}\ (S_i, S_j) := \\
&\qquad \exists\, r\!: \text{SL\_resource}. \\
&\qquad\quad \text{Stage\_Type}\ (S_i) \neq \text{Stage\_Type}\ (S_j)\ \wedge \\
&\qquad\quad \text{Stage\_Used}\ (S_i, r) \wedge \text{Stage\_Used}\ (S_j, r)
\end{aligned}$$

Looking closer, even when the predicate Dual_Stage_Conflict is true, a conflict occurs only if the stage instructions S_i and S_j use the resource r at the same phase of the clock, since a multi-phased non-overlapping clock is used (figure 3).

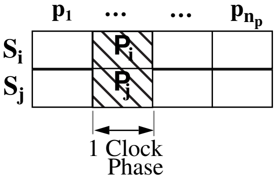

Figure 3: Phase Parallel Execution

Having an implementation of the stage instructions at the phase level and considering all combinations of phase instructions for any two stage instructions, the dual stage conflict is defined at this lower level in terms of a multiple phase conflict predicate. Formally, Multiple_Phase_Conflict is defined as disjunctions over all possible phase instruction pair conflicts. Let Dual_Phase_Conflict be a predicate representing dual phase conflicts, the dual stage conflict is specified formally as follows (where according to the index k, ϕ_k represents a clock phase, i.e. ϕ_1 = Ph1, ϕ_2 = Ph2):

$$\begin{aligned}
&\text{Dual\_Stage\_Conflict} = \\
&\quad \vdash_{def}\ \text{Multiple\_Phase\_Conflict}\ (S_i, S_j) := \\
&\quad \bigvee_{\substack{k \\ (k = 1 \ldots n_p)}} \text{Dual\_Phase\_Conflict}\ (\text{StageToPhase}\ (\phi_k, S_i), \\
&\qquad\qquad\qquad\qquad\qquad \text{StageToPhase}\ (\phi_k, S_j))
\end{aligned}$$

Phase Level Conflicts. A dual resource conflict at the phase level occurs only when any two phase instructions that compete for the same resource, are of the same phase type, i.e. the same clock phase is involved (see figure 3) and belong to stage instructions of different types. Using the functions Phase_Type, Stage_Type and StageToPhase and the predicate Phase_Used, this is formally given as follows:

$$\begin{aligned}
&\vdash_{def}\ \text{Dual\_Phase\_Conflict}\ (P_i, P_j) := \\
&\qquad \exists\, r\!: \text{PL\_resource}. \\
&\qquad\quad \text{Phase\_Type}\ (P_i) = \text{Phase\_Type}\ (P_j) \wedge \\
&\qquad\quad (\text{Stage\_Type}\ (\text{PhaseToStage}\ (P_i)) \neq \\
&\qquad\quad\ \text{Stage\_Type}(\text{PhaseToStage}(P_j))) \wedge \\
&\qquad\quad \text{Phase\_Used}\ (P_i, r) \wedge \text{Phase\_Used}\ (P_j, r)
\end{aligned}$$

5.2 Resource Conflict Proof

Our ultimate goal is to show that for all class instruction combinations, no resource conflicts occur, i.e. the predicate Multiple_Res_Conflict is never true:

$$``\forall I_1 \ldots I_{n_s} : \text{class\_instruction} . \quad \neg \text{ Multiple\_Res\_Conflict y } (I_1, \ldots, I_{n_s})"$$

Using the definition of Multiple_Res_Conflict (cf. section 5.1), the expansion of this goal at the stage level yields to a case explosion. In order to bypass this intractable goal, we do the proof in two steps as follows:

1. we prove that dual conflicts cannot occur:

$$``\forall\ S_i\ S_j : \text{stage\_instruction} . \quad \neg \text{ Dual\_Stage\_Conflict } (S_i, S_j)"$$

2. we conclude the negation of the multiple conflict predicate from the first step:

$$``(\forall\ S_i\ S_j : \text{stage\_instruction} . \ \neg \text{ Dual\_Stage\_Conflict } (S_i, S_j)) \Rightarrow (\forall I_1 \ldots I_{n_s} : \text{class\_instruction}. \ \neg \text{ Multiple\_Res\_Conflict } (I_1, \ldots, I_{n_s}))"$$

Since the dual conflict predicate is a generalization of the multiple conflict predicate, the proof of the second step is straightforward; we even do not need to expand the dual conflict predicate. The proof of the first step, without any assumptions, leads either to *True,* or to a number of subgoals which explicitly include a specific resource and the specific stage instructions which conflict. For example a conflict due to the resource PC between the common IF-stage instruction (IF_X) and the ID-stage instruction (ID_C) of control class is output as follows:

$$(S_i = \text{IF\_X}), (S_j = \text{ID\_C}), (r = \text{PC}) \vdash \texttt{F}$$

Referring to the last example, the simultaneous use of the resource PC at the phase level is checked by explicitly setting the following goal using the Multiple_Phase_Conflict predicate:

$$``\neg \text{ Multiple\_Phase\_Conflict (IF\_X, ID\_C, PC)}"$$

If the goal is proven to be true, then the conflict freedom is shown, else the resource conflict remains and in order to resolve it, the implementation EBM has to be changed appropriately, e.g. by using an additional buffer.

6 Data Conflicts

Data conflicts (also called *data hazards* [8, 11, 15], *timing hazards* [13], *data dependencies* [7] or *data races* [5]) arise when an instruction depends on the results of a previous instruction. Such data dependencies could lead to faulty computations when the order in which the operands are accessed is changed by the pipeline.

Data conflicts are of three types called, read after write (RAW), write after read (WAR) and write after write (WAW) [8, 11, 15, 7] (also called destination source (DS), source destination (SD) and destination destination (DD) conflicts [13]). Given that an instruction I_j is issued after I_i, a brief description of these conflicts is:

RAW conflict – I_j *reads* a source before I_i *writes* it
WAR conflict – I_j *writes* into a destination before I_i *reads* it
WAW conflict –I_j *writes* into a destination before I_i *writes* it

6.1 Data Conflict Specification

Data conflicts include temporal aspects that are related to the temporal abstractions of our hierarchical model. Therefore, similar to resource conflicts, the formal specifications of data conflicts are considered at different abstraction levels.

Class Level Conflicts. Considering a full pipeline (see figure 2), the data conflict predicate, i.e. Data_Conflict, should involve the maximal number n_s of instructions that could lead to data conflicts. The predicate Data_Conflict is defined in terms of a multiple data conflict predicate, which includes n_s instructions $I_1 \ldots I_{n_s}$ with corresponding sequential issue times t_{o1} … t_{on_s}[4]. The predicate Multiple_Data_Conflict is true whenever any two class instructions conflict on some data. Hence, we define Multiple_Data_Conflict as the disjunction of all possible dual data conflicts (represented by Dual_Data_Conflict) as follows:

$$\begin{array}{l} \text{Data\_Conflict} = \\ \quad \vdash_{def} \text{ Multiple\_Data\_Conflict } (I_1, \ldots, I_{n_s}) := \\ \qquad \exists\ t_{o1} \ldots t_{on_s} : \text{time} . \\ \qquad \displaystyle\bigvee_{\substack{i,j \\ (i,j = 1 \ldots n_s) \\ (i<j)}} \text{Dual\_Data\_Conflict } ((I_i, t_{oi}), (I_j, t_{oi}+j-1)) \end{array}$$

The predicate Dual_Data_Conflict is true, if there exists a resource of the programming model (class level) for which two class instructions I_i and I_j issued at time points t_{oi} and t_{oj}, respectively, conflict. Further, according to our hierarchical model, the Dual_Data_Conflict is handled hierarchically, first at the stage then at the phase level. Formally, Dual_Data_Conflict is defined in terms of a Stage_Data_Conflict predicate, as follows:

$$\begin{array}{l} \vdash_{def} \text{ Dual\_Data\_Conflict } ((I_i, t_{oi}), (I_j, t_{oj})) := \\ \qquad \exists\ r : \text{CL\_resource} . \\ \qquad\qquad \text{Stage\_Data\_Conflict } ((I_i, t_{oi}), (I_j, t_{oj}), r) \end{array}$$

Stage Level Conflicts. Let I_i be an instruction that is issued into the pipeline at time t_{oi} and *writes* a given resource r at t_{ui} ($t_{oi} \le t_{ui}$). Let I_j be another instruction that is issued at later time t_{oj}, i.e. ($t_{oi} < t_{oj}$) and *reads* the same resource r at t_{uj}. A RAW data conflict occurs when the resource r is read by I_j *before* (and not *after*) this resource is written by the previous sequential instruction I_i (figure 4). Let s_i and s_j be the related pipeline stages in which the resource r is written and read, respectively. Let θ be a symbol representing the function Stage_Rank, which computes the ordinal value of a given pipeline stage (cf. section 4). Assuming a linear pipeline execution of instructions, i.e. no pipeline freeze or

---

[4] We assume a linear pipelining of instructions, i.e. no pipeline freeze or stall exist, as far as data conflicts are concerned. Stalls are handled as control conflicts, as described in section 7.2.

stall happen, the use time points t_{ui} and t_{uj} are equal to ($t_{oi} + \theta(s_i)$) and ($t_{oj} + \theta(s_j)$), respectively. Hence, the timing condition for the RAW conflict, i.e. ($t_{uj} \le t_{ui}$), is equivalent to ($t_{oj} - t_{oi}) \le (\theta(s_i) - \theta(s_j)$).

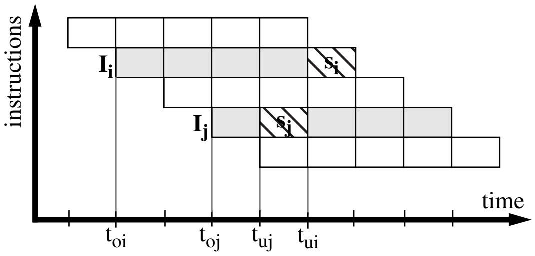

Figure 4: RAW Data Conflict

Using the function Stage_Rank (represented by the symbol θ) and the predicates Stage_Range and Stage_Domain, the formal specification of the stage RAW data conflict is thus given as follows:

$\vdash_{def}$ Stage_RAW_Conflict $((I_i, t_{oi}), (I_j, t_{oj}), r)$:=
$\exists\, s_i\, s_j$: pipeline_stage.
$(0 < (t_{oj} - t_{oi})) \wedge (t_{oj} - t_{oi}) \le (\theta(s_i) - \theta(s_j))) \wedge$
Stage_Range $(I_i, s_i, r) \wedge$ Stage_Domain (I_j, s_j, r)

Similarly, the WAR and WAW predicates are defined as follows, where the semantics of the data conflict is reflected by the order of the Stage_Range and Stage_Domain predicates:

$\vdash_{def}$ Stage_WAR_Conflict $((I_i, t_{oi}), (I_j, t_{oj}), r)$:=
$\exists\, s_i\, s_j$: pipeline_stage.
$(0 < (t_{oj} - t_{oi})) \wedge (t_{oj} - t_{oi}) \le (\theta(s_i) - \theta(s_j))) \wedge$
Stage_Domain $(I_i, s_i, r) \wedge$ Stage_Range (I_j, s_j, r)

$\vdash_{def}$ Stage_WAW_Conflict $((I_i, t_{oi}), (I_j, t_{oj}), r)$:=
$\exists\, s_i\, s_j$: pipeline_stage.
$(0 < (t_{oj} - t_{oi})) \wedge (t_{oj} - t_{oi}) \le (\theta(s_i) - \theta(s_j))) \wedge$
Stage_Range $(I_i, s_i, r) \wedge$ Stage_Range (I_j, s_j, r)

Phase Level Conflicts. A special case of the data conflict timing condition arises when a resource is simultaneously used by the instructions I_i and I_j, i.e. $t_{ui} = t_{uj}$. In this situation, the data conflict should be examined at the phase level. Let S_i and S_j be any two stage instructions, where the rank of S_i is greater than that of S_j, e.g. S_i = WB_L and S_j = ID_C. According to figure 5, a RAW data conflict at the phase level happens when the resource r is *written* by the stage instruction S_i at a clock phase p_i that occurs *after* clock phase p_j, where it is *read* by S_j, i.e. ($\tau_{ui} \ge \tau_{uj}$). Since instructions at the phase level are executed purely in parallel, they all have the same issue time τ_o (figure 5), the timing condition ($\tau_{ui} \ge \tau_{uj}$) is equivalent to ($\tau_o + \xi(p_i)) \ge \tau_o + \xi(p_j)) = (\xi(pi) \ge \xi(p_j)$), where the symbol ξ represents the function Phase_Rank which computes the ordinal value of the clock phase (cf. section 4). Using the functions Stage_Rank, Phase_Rank and Stage_Type and the predicates Phase_Domain and Phase_Range, the phase level RAW data conflict predicate is formally given as follows (where the symbol ϑ represents the function Stage_Type which computes the logical stage type of a stage instruction (cf. section 4):

$\vdash_{def}$ Phase_RAW_Conflict (S_i, S_j, r):=
$\exists\, p_i\, p_j$: clock_phase.
$(\xi\,(p_i) > \xi\,(p_j)) \wedge (\theta(\vartheta(S_i)) > (\theta(\vartheta\,(S_j)) \wedge$
Phase_Range $(S_i, p_i, r) \wedge$ Phase_Domain (S_j, p_j, r)

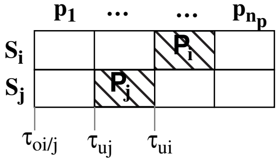

Figure 5: Phase Level RAW Data Conflict

In a similar manner, the formal definitions of the phase level WAR and WAW data conflict predicates are given as follows:

$\vdash_{def}$ Phase_WAR_Conflict (S_i, S_j, r):=
$\exists\, p_i\, p_j$: clock_phase.
$(\xi\,(p_i) > \xi\,(p_j)) \wedge (\theta(\vartheta(S_i)) > (\theta(\vartheta\,(S_j)) \wedge$
Phase_Domain $(S_i, p_i, r) \wedge$ Phase_Range (S_j, p_j, r)

$\vdash_{def}$ Phase_WAW_Conflict (S_i, S_j, r):=
$\exists\, p_i\, p_j$: clock_phase.
$(\xi\,(p_i) > \xi\,(p_j)) \wedge (\theta(\vartheta(S_i)) > (\theta(\vartheta\,(S_j)) \wedge$
Phase_Range $(S_i, p_i, r) \wedge$ Phase_Range (S_j, p_j, r)

6.2 Data Conflict Proof

Our ultimate goal in proving the non existence of data conflicts relies in showing that none of the data conflicts (RAW, WAR and WAW) occurs. This proof is split into three independent parts each corresponding to one data conflict type. These proofs are similar and in the following we will handle RAW conflicts for illustration purposes.

At the top-most level, the goal to be proven for RAW data conflicts is given in terms of the multiple RAW data conflict predicate as follows:

"$\forall\, I_1 \ldots\, I_{n_s}$: class_instruction.
$\neg$ Multiple_RAW_Conflict $(I_1, \ldots, I_{n_s})$"

This goal includes a quantification over all possible conflict combinations that could occur between all permutations of n_s instruction within the pipeline. As in the case of resource conflicts, the direct proof of this goal yields to a case explosion. Hence, we manage the proof in two steps in that we first prove that the dual conflicts do not occur and then conclude the negation of the multiple conflict predicate from the first step. The proof of step 2 is done in a straightforward manner. The goal for the first step is given as follows:

"$\forall\, I_i\, I_j$: class_instruction.
$\forall\, t_{oi}\, t_{oj}$: time.
$\forall\, r$: CL_Resource.
$\neg$ Stage_RAW_Conflict $((I_i, t_{oi}), (I_j, t_{oj}),$ r)"

The proof of this goal yields either *True* or a number of subgoals, which include the specific resource and class instructions that conflict. For example, a data conflict that occurs between LOAD and ALU-instructions due to the resource register file RF, which is written at the WB-stage by the LOAD-instruction and read at the ID-stage by the ALU-instruction is detected and output as follows, where the number "3" corresponds to the difference $\theta(s_i) - \theta(s_j)$ = "$\theta(\text{WB}) - \theta(\text{ID})$":

$$\begin{array}{l} (I_i = \text{LOAD}), (I_j = \text{ALU}), (s_i = \text{WB}), \\ (s_j = \text{ID}), (r = \text{RF}), (0 < (t_{oj} - t_{oi})) \end{array} \vdash \neg ((t_{oj} - t_{oi}) \leq 3)$$

This result is interpreted as follows: as long as the issue times of the conflicting LOAD and ALU-instructions satisfy the condition "$(t_{oj} - t_{oi}) \leq 3$", there exists a data conflict. In order to resolve this conflict, we should neutralize this timing condition. This can be done by considering the following two cases:

1. "$(t_{oj} - t_{oi}) = 3$": the data conflict should be explored at the lower time granularity, i.e. phase level, by setting the following goal:

$$\text{"}\neg\ \text{Phase\_RAW\_Conflict}\ (\begin{array}{l} \text{ClassToStage (WB, LOAD)}, \\ \text{ClassToStage (ID, ALU)}, \\ \text{RF)"} \end{array}$$

If the goal is proven correct, no data conflict happens, else either the hardware EBM should be changed, e.g. via the inclusion of more clock phases, or one uses the software scheduling technique [8].

2. "$(t_{oj} - t_{oi}) < 3$": The timing information gives an exact reference for the maximum number of pipeline slots or bypassing paths that have to be provided by the software scheduling technique or the implementation EBM, respectively, namely (3-1 = **2**).

Using instruction scheduling, the needed software constraint (assumption) that leads to the proof of the dual data conflict goal, could then be defined as:

$$\begin{array}{l} \vdash_{def}\ \text{SW\_Constraint:=} \\ \qquad (I_i = \text{LOAD}) \wedge (I_j = \text{ALU}) \wedge \\ \qquad (r = \text{RF}) \wedge (0 < (t_{oj} - t_{oi})) \quad \Rightarrow \quad ((t_{oj} - t_{oi}) > 3) \end{array}$$

Using the *bypassing* (also called *forwarding*) technique [8], we ensure that the needed data is forwarded as soon as it is computed (end of the EX-stage) to the next instruction (beginning of the EX-stage). This behaviour is implemented in hardware by using some registers and corresponding feedback paths that hold and forward this data, respectively. The existence of these registers and corresponding forward paths can be easily formalized as follows:

$$\begin{array}{l} \vdash_{def}\ \text{Bypassing:=} \\ \qquad \exists\, rb.\ \ (rb = \text{RF}) \wedge \\ \qquad\qquad \text{Stage\_Range}\ (I_i, \text{EX}, rb) \wedge \\ \qquad\qquad \text{Stage\_Domain}\ (I_j, \text{EX}, rb) \end{array}$$

Having shown this bypassing condition from the implementation EBM[5] and assuming it in the dual data conflict goal, the existentially quantified pipeline stage variables s_i and s_j in the definition of Stage_RAW_Conflict (cf. section 6.1) are set to EX and the timing condition is hence reduced to:

$$\ldots, (0 < (t_{oj} - t_{oi})) \vdash \neg ((t_{oj} - t_{oi}) \leq 0)$$

which is always true.

To summarize, given some specific software constraints in form of instruction scheduling timing conditions and given the implementation EBM, which includes some bypassing paths with appropriate logic, we are able to prove that none of the data conflicts (RAW, WAR and WAW) happens, i.e. formally:

$$\text{SW\_Constraints, EBM} \vdash \begin{array}{l} (\neg\ \text{RAW\_Data\_Conflict} \wedge \\ \neg\ \text{WAR\_Data\_Conflict} \wedge \\ \neg\ \text{WAW\_Data\_Conflict}) \end{array}$$

7 Control Conflicts

Control conflicts (also called *control hazards* [8, 15], *branch hazards* [8], *sequencing hazards* [13] or *branch dependencies* [7]) arise from the pipelining of branches and other instructions that change the program counter *PC*, i.e. interruption of the linear instruction flow.

Control conflicts involve all kinds of interrupts to the linear pipeline flow. This includes the software branches, jumps, traps, etc. and hardware interrupts, stalls, etc. Using this classification, we define *SW control conflicts* and *HW control conflicts* as those conflicts which are caused by a software instruction and a hardware event, respectively. The correctness proof of *Control_conflict* is therefore split into two independent parts as follows:

7.1 SW Control Conflict

Let $A_f(I_i, t_{oi})$ be the fetch address of an instruction I_i issued at the time point t_{oi} and $A_n(I_i, t_{oi})$ be the fetch address of the next instruction (also called next address of I_i). Let *PC* be the program counter. In a pipelined instruction execution, at each clock cycle a new instruction is fetched, i.e. $\forall\ t.\ PC(t+1) = Inc\ PC(t)$[6]. The fetch and next addresses of an instruction I_i are equal to $A_f(I_i, t_{oi}) = PC(t_{oi})$ and $A_n(I_i, t_{oi}) = PC(t_{oi}+1)$, respectively. During a sequential program execution, we have $A_n(I_i, t_{oi}) = A_f(I_i+1, t_{oi}+1)$ and we obtain the following:

$$A_n(I_i, t_{oi}) = A_f(I_i+1, t_{oi}+1) = PC(t_{oi}+1) = Inc\ PC(t_{oi})$$

If I_i is a control instruction, then the next address is its target address, i.e. $A_n(I_i, t_{oi}) = A_t(I_i, t_{oi})$. Due to the sequential execution of a single instruction, the target instruction can only be fetched after the instruction I_i is fetched, decoded and the target address has been calculated. Since all this cannot happen in one clock cycle, the target address $A_t(I_i, t_{oi})$ is equal to $PC(t_{oi}+n)$, where $n > 1$. Hence, the next address is not equal to the target address, i.e.:

$$A_n(I_i, t_{oi}) = Inc\ PC(t_{oi}) \neq A_t(I_i, t_{oi}) = PC(t_{oi}+n)$$

and the wrong instruction is fetched next.

A closer look at this situation shows that a software control conflict occurs when an instruction attempts to read

---

[5] Since the implementation already includes in hardware the required buffers and bypass paths, this is done easily through a manual extraction from EBM.

[6] *Inc* is an increment function which is independent of the increment value, e.g. PC+4, PC+8, etc.

the resource *PC* that is not yet updated (written) by a previous instruction. This complies with the definition of RAW data conflict in *PC* [11] and the software control conflict could be defined as follows:

$$\vdash_{def} \text{SW\_Control\_Conflict} := \text{Stage\_RAW\_Conflict}\ ((\text{CONTROL}, t_{oi}), (I_j, t_{oj}), \text{PC})$$

The conflict freedom proof is therefore only a special case of the data conflict proofs and the goal to be proven is set as follows:

$$\begin{array}{l} \text{"}\forall\, I_j : \text{class\_instruction}\,. \\ \quad \forall\, t_{oi}\, t_{oj} : \text{time}\,. \\ \quad \neg\, \text{Stage\_RAW\_Conflict}\ ((\text{CONTROL}, t_{oi}), (I_j, t_{oj}), \text{PC})\text{"} \end{array}$$

and for the DLX processor example, we obtain 5 subgoals of the following form:

$$\begin{array}{l} (I_j = \text{ALU}), (s_i = \text{ID}), \\ (s_j = \text{IF}), (0 < (t_{oj} - t_{oi})) \end{array} \vdash \neg\, ((t_{oj} - t_{oi}) \leq 1)$$

For conflict resolution no bypassing is possible, since the calculation of the target address cannot be done earlier. The commonly used technique is software scheduling [8]. In the DLX RISC processor, we just need one delay slot ($(t_{oj} - t_{oi}) \leq 1$) to ensure that control instructions are executed correctly. The given software constraint that is used in this case could be defined as follows:

$$\vdash_{def} \text{SW\_Constraint} := (0 < (t_{oj} - t_{oi})) \Rightarrow ((t_{oj} - t_{oi}) > 1)$$

7.2 HW Control Conflicts

In contrast to the previous conflict kind, this kind of conflict cannot be handled in general since the interrupt behaviour and conflict resolution are fully hardware implementation dependent. Different RISC processors handle interrupts, stalls, freezes and branch prediction in different ways, i.e where the forced branch is to be inserted, how to restart exactly the interrupted program, etc.

In general, one has to manually specify the hardware (interrupts, stalls, freezes, branch prediction) behaviour in the form of a predicate, whose implication has to be proven correct from the implementation EBM, e.g. for hardware interrupts:

$$\vdash \text{EBM} \Rightarrow \text{INTR\_SPEC}$$

The predicate INTR_SPEC[7] describes the behaviour of the implemented hardware interrupt and ensures that no conflict occurs when the linear pipeline flow is interrupted, e.g. proper saving and recovery of the processor state before and after the interrupt handling.

Having implied from the implementation EBM the intended behaviours of all processor specific hardware control mechanisms, we obtain:

$$\text{EBM} \vdash \neg\, \text{HW\_Control\_Conflict}$$

and consequently for both the SW and HW control conflicts:

$$\text{SW\_Constraints, EBM} \vdash \begin{array}{l} (\neg\, \text{SW\_Control\_Conflict} \wedge \\ \ \neg\, \text{HW\_Control\_Conflict}) \end{array}$$

Summary

Our ultimate goal in proving the non-existence of pipeline conflicts is concluded from the theorems yielded in sections 5.2, 6.2, 7.1 and 7.2, i.e.:

$$\text{SW\_Constraints, EBM} \vdash \begin{array}{l} (\neg\, \text{Res\_Conflict} \wedge \\ \ \neg\, \text{Data\_Conflict} \wedge \\ \ \neg\, \text{Control\_Conflict}) \end{array}$$

We have been able to automate most of this process, however, manual specifications which are implementation dependent are necessary to tackle the control conflicts which are circumvented using hardware.

8 Experimental Results

The methodology presented so far, has been validated by using the DLX processor as an example. This processor contains a 5 stage pipeline with a 2 phased clock, and its architecture includes 51 basic instructions (integer, logic, load/store and control). All these instructions are grouped into 4 classes according to which the stage and phase instructions are defined (refer to table 1). The EBM for the DLX core has been implemented in a commercial VLSI design environment using a 1.0 m CMOS technology. The design has approximately 150,000 transistors which occupies a silicon area of about 60.34 mm^2, it has 172 I/O pads and currently runs at a clock rate of 12.5 MHz [4].

The formalization of the specifications for DLX have been done within the HOL verification system [6]. The implementation of the interpreter model specifications and the proof of semantic correctness is reported in [18]. The overall specification for the DLX core is about 4760 lines long and the proof of the semantic step took about 457.66 secs on a SPARC10, with a 96 MB main memory.

In proving the pipeline correctness, the specification and tactics for each kind of conflict have been implemented in HOL90.6. These specifications and tactics are general enough to be applicable to a wide range of other RISC processors. The hierarchical structuring of the proofs resulted in parameterized tactics that are used for more than one kind of conflict. All proofs have been mainly done using 4 proof tactics — MULTIPLE_CONF_TAC, RES_CONF_TAC, DATA_CONF_TAC and CONTROL_CONF_TAC — with a total proof script text of 850 lines. The overall definition and specification text for the pipeline corectness is about 2690 lines. The run times, on a SPARC10 with a 96 MB main memory, for the theorem generation of the Used and Range/Domain predicates and for the pipeline verification, respectively, are given for the DLX processor example in the following table:

---

[7] A formal specification of INTR_SPEC for DLX is beyond the scope of this paper and is not presented here.

Table 2: Verification Results for the DLX processor

Verification Goal	Time (in sec)	Comments
Stage_Used Gen.	247.27	208 theorems generated
Phase_Used Gen.	1718.03	422 theorems generated
Stage Ran. & Dom. Gen.	324.44	200 theorems generated
Phase Ran. & Dom. Gen.	272.27	260 theorems generated
Stage Dual Res. Conf.	1691.64	0 conflicts
Dual ⇒ Multiple (Res.)	1.30	-
Stage RAW Data Conf.	570.64	8 conflict cases (3 slots) by RF and 4 conflict cases (1 slot) by PC
(using Bypassing)	1.89	0 conflicts
(using SW-scheduling)	458.96	0 conflicts
Stage WAR Data Conf.	572.23	0 conflicts
Stage WAW Data Conf.	572.50	0 conflicts
Dual ⇒ Multiple (Data)	0.27	-
SW Control Conflicts	32.44	4 conflict cases (1 slot)
Σ Pipeline Correctness	6464.18	–

9 Conclusions and Future Work

In this paper, we have described the formalization and the verification of the different kinds of conflicts that occur in the instruction pipelines of RISC cores. The employment of the hierarchical RISC interpreter model and in particular the exploitation of the class level, empowers us to automatically derive compact specifications of the conflicts. We have implemented constructive proofs for these conflicts and hence the designer gets invaluable feedback for resolving these conflicts, either by making appropriate modifications to the hardware or by generating the required software constraints.

The interpreter model, the formal specifications and the proof techniques are generic and hence applicable to any RISC core. Furthermore, the RISC interpreter model closely reflects the RISC design hierarchy thus the specifications in higher-order logic can be more or less automated. Given such specifications and an implementation, the proof process has been automated by using parametrizable tactics. These tactics are independent of the underlying implementation (except in the case of hardware control conflicts) and can be used for a large number of RISC cores. The entire methodology has been validated by using the DLX core.

The feasibility of formal verification techniques, when applied cleverly to specific classes of circuits is illustrated by the runtimes shown in table 2. In our future work, we shall extend the layer of the core to include pipelined functional units, floating point processor, etc.

References

1. Anceau, F.: The Architecture of Microprocessors; Addison-Wesley Publishing Company, 1986.
2. Buckow, O.: Formale Spezifikation und (Teil-) Verifikation eines SPARC-kompatiblen Prozessors mit LAMBDA; Diplomarbeit, Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, Oktober 1992.
3. Cohn, A.: A Proof of the Viper Microprocessor: The First Level; In: Birtwistle, G. and Subrahmanyam, P. (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988.
4. Dehof, M.; Tahar, S.: Implementierung des DLX RISC-Prozessors in einer Standardzellen-Entwurfsumgebung; Technical Report No. SFB 358-C2-1/94, Institute for Computer Design and Fault Tolerance, University of Karlsruhe, Germany, March 1994.
5. Furber, S.: VLSI RISC Architecture and Organization; Electrical Engineering and Electronics, Dekker, New York, 1989.
6. Gordon, M.; Melham, T.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic; Cambridge, University Press, 1993.
7. Van De Goor, A.: Computer Architecture and Design; Addison-Wesley, 1989.
8. Hennessy, J.; Patterson, D.: Computer Architecture: A Quantitative Approach; Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
9. Hunt, W.: Microprocessor Design Verification; Journal of Automated Reasoning, Vol. 5, 1989, pp. 429-460.
10. Joyce, J.: Multi-Level Verification of Microprocessor-Based Systems; Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989.
11. Kogge, P.: The Architecture of Pipelined Computers; McGraw-Hill, 1981.
12. 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, No. 2, 1993, pp. 165-230.
13. Milutinovic, V.: High Level Language Computer Architecture; Computer Science Press, Inc., 1989.
14. Srivas, M.; Bickford, M.: Formal Verification of a Pipelined Microprocessor; IEEE Software, September 1990, pp. 52-64.
15. Stone, H.: High-Performance Computer Architecture; Addison-Wesley Publishing Company, 1990.
16. Tahar, S.; Kumar, R.: A Formalization of a Hierarchical Model for RISC Processors; In: Spies, P. (Ed.), Proc. European Informatics Congress Computing Systems Architecture, Munich, October 1993, Informatik Aktuell, Springer Verlag, pp. 591-602.
17. Tahar, S.; Kumar, R.: Towards a Methodology for the Formal Hierarchical Verification of RISC Processors; Proc. IEEE International Conference on Computer Design, Cambridge, Massachusetts, October 1993, IEEE Computer Society Press, pp. 58-62.
18. Tahar, S.; Kumar, R.: Implementing a Methodology for Formally Verifying RISC Processors in HOL; In: Joyce, J.J. and Seger, C. (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 281-294.
19. Tahar, S.; Kumar, R.: Formal Verification of Pipeline Conflicts in RISC Processors; Proc. EURODAC-94, Grenoble, France, September 1994, IEEE Computer Society Press, pp. 285-289.
20. Windley, P.: Formal Modeling and Verification of Microprocessors; IEEE Transactions on Computers, Vol. 44, No.1, 1995.