EURO-DAC'94
European Design Automation Conference
Grenoble, France, September 1994
IEEE Computer Society Press, pp. 285-289

Formal Verification of Pipeline Conflicts in 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

We outline a general methodology for the formal verification of pipeline conflicts in RISC cores. The different kinds of conflicts that can occur due to the simultaneous execution of the instructions in the pipeline have been formalized and automated proof techniques for each kind of conflict have been given. When conflicts are detected during the proof process, the conditions under which these occur are generated, thus aiding a designer in their removal. The described formalizations and proofs have been illustrated via the DLX RISC processor.

1. Introduction

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 past work in formally verifying microprocessors has mostly concentrated on the verification of microprogrammed processors. Although large examples have been verified [2, 5] and a general methodology for verifying microprogrammed processors has been given [6, 13], these efforts do not reflect the complexity of the commercially available CISC processors. We have therefore focused our efforts on developing a methodology for the verification of RISC cores, since they have simpler instruction sets.

The previous work in the verification of RISC processors were only able to verify parts of processors at certain levels of abstraction [9]. 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 [10]. Using this model, the 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 [11]. Parts of the formal proofs which do not deal with the conflicts occurring due to pipelining, have been described in [12].

In this paper, we focus our attention on the formalization and proofs of pipeline conflicts. These conflicts occur due to the data and control dependencies and resource contentions when many instructions are simultaneously executed in the pipeline. We have formalized all possible conflicts and described largely automated proof techniques for conflict detections. The proof techniques that are given are constructive, i.e. the conditions under which the conflicts occur are explicitly stated, so that the designer can easily formulate the conflict resolution mechanisms either in hardware or generate software constraints which have to be met.

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 adapted. Sections 4 through 6 define the conflicts arising due to the pipelined nature of RISCs and describe the correctness proofs for resource, data and control conflicts, respectively. Section 7 contains some experimental results and section 8 concludes the paper. It is to be noted, that all the methods and techniques presented in this paper are illustrated by means of a RISC example – DLX[1] [4].

2. RISC Verification Model

Our RISC interpreter 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 Joyce [6] and Windley [13]. Instead of directly showing that the implemented circuit (Electronic Block Model – EBM) correctly implements each instruction, they have shown the correctness between the neighbouring abstraction levels and thus reduced the complexity of the verification process. Although the RISC processors also possess similar

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.

levels of hierarchy in the design, a naive mapping of this interpreter model onto RISCs does not reduce the complexity of the verification process, as shown in [10].

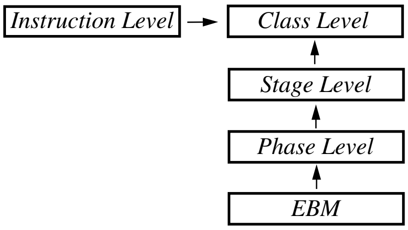

Figure 1: RISC Interpreter Model

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 which corresponds to the circuit implementation – EBM (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 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 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 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.

In our previous papers, we have given a detailed formalization of instructions at all abstraction levels [10, 11, 12]. We briefly recapitulate some of the needed definitions here. Table 1 shows the pipeline structure of DLX. The columns represent the four instruction classes – ALU, LOAD, STORE and CONTROL. The rows correspond to the related stage and phase transfers at the five pipeline stages – IF, ID, EX, MEM and WB (Type: *pipeline_stage*) and the two clock phases ϕ_1 and ϕ_2 (Type: *clock_phase*), respectively. The stage and phase instructions are directly derived from the pipeline structure, e.g. the stage instruction ID_C (for the instruction class CONTROL and the pipeline stage ID) is defined in terms of the transfers "$BTA \leftarrow f_c(PC)$" and "$PC \leftarrow BTA$". Furthermore, we define at each abstraction level an enumeration instruction type, e.g. the type *Class_Instruction:= ALU | LOAD | STORE | CONTROL* for the class level. We have also defined types of resources which are related to the structural abstraction (seen storage elements), e.g. at the class level – *Resource:= PC | RF | I_MEM* | ... The sequential order of the execution of the stage and phase instructions is indicated by the ordinal value of the related pipeline stage or clock phase, respectively, e.g. $IF = 1, ID = 2,$ etc. or $\phi_1 = 1, \phi_2 = 2,$ etc.

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. Formal Verification Process

In formally verifying a RISC processor, we have to show that the instruction set is executed correctly by the EBM, in spite of the pipelined architecture. Given that n_s is the pipeline depth, a RISC processor executes n_s instructions in parallel (see figure 2), in n_s different pipeline stages. Although the overall throughput of the processor is increased, no single instruction runs faster. Hence we have to prove that the sequential execution of each instruction is correctly implemented by the EBM. Furthermore, the existence of data/control dependencies and the resource contentions between the instructions in the pipeline could lead to semantical inconsistencies. Therefore, we have to prove that the pipelined sequencing of instructions is correctly implemented. 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.:

   $SW\_Constraints, EBM \vdash Correct\_Instr\_Pipelining$ **(1)**

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

   $\vdash EBM \Rightarrow Instruction\ Level$ **(2)**

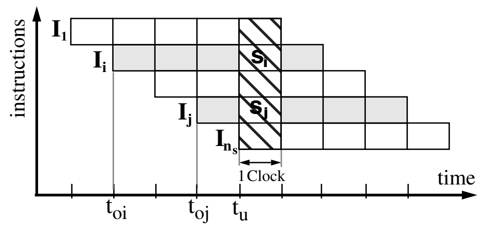

Figure 2: Pipelined Execution

The software constraints in (1) are 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.

Step (2), has been discussed and reported in our previous work [11, 12] and we recapitulate it briefly in the next subsection. Step (1) is the main topic of this paper and will be discussed in detail in the rest of the paper.

3.1 Semantical Correctness

In order to show that the sequential execution of each instruction is correctly implemented by the EBM, we use the higher-order logic specifications and implementations at the various levels of abstraction and prove the following – EBM ⇒ Phase Level ⇒ Stage Level ⇒ Class Level. This process is similar to the one used by Windley in proving the correctness of microprogrammed processors [13]. Later, these proofs are instantiated for each instruction at the instruction level. We have implemented proof scripts in the HOL theorem prover [3], using the hardware verification environment – MEPHISTO [8], which automates the entire process, given the formal definitions of the specifications and implementations at each abstraction level.

3.2 Pipeline Correctness

The correctness of step (1) consists in the proof that all possible combinations of n_s instructions are executed correctly. This is equivalent to the proof that at any time, 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 namely, resource, data and control conflicts [4]. Since the pipeline correctness is the direct consequence of the absence of all these conflicts, the correctness statement (1) defines the non-existence of these conflicts. The predicate *Correct_Instr_Pipelining* in (1) is hence defined as the following conjunction, where we assign to each kind of conflict a specific conflict predicate, i.e. *Resource_Conflict*, *Data_Conflict* and *Control_Conflict*. Formally:

$$\begin{aligned} \mathit{Correct\_Instr\_Pipelining} := \; & (\neg\, \mathit{Resource\_Conflict} \wedge \\ & \neg\, \mathit{Data\_Conflict} \wedge \\ & \neg\, \mathit{Control\_Conflict}) \end{aligned}$$

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

$$\mathit{SW\_Constraints},\ \mathit{EBM} \;\vdash\; \begin{aligned} & (\neg\, \mathit{Resource\_Conflict} \wedge \\ & \neg\, \mathit{Data\_Conflict} \wedge \\ & \neg\, \mathit{Control\_Conflict}) \end{aligned}$$

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.

The proof of the predicate *Correct_Instr_Pipelining* ensures that all possible combinations of instructions that occur in the n_s pipeline stages are executed correctly. Exploiting the notion of class instructions, as described in section 2, the case analysis explosion is avoided by considering the combinations of few classes instead of combinations of all instructions. All conflict predicates are closely related to our hierarchical interpreter model and will be specified at a higher level in terms of class instructions, taking the temporal and structural abstractions into account.

4. Resource Conflicts

Resource conflicts occur when some resources are not duplicated enough and two or more instructions attempt to use them simultaneously [4, 7]. A resource could be a register, a memory unit, a functional unit, a bus, etc. At any time the RISC processor executes up to n_s instructions simultaneously, where each instruction is in a *different* pipeline stage. A resource conflict occurs whenever *two or more* stage instructions of the n_s stage instructions, which are in parallel execution (see hatched box in figure 2), attempt to use the *same* resource. The use of a resource is a write operation for storage elements and an allocation for functional units.

4.1 Resource Conflict Specification

Let I_i and I_j be two sequential instructions within the pipeline that are issued at times t_{oi} and t_{oj}, respectively ($0 < (t_{oj} - t_{oi}) < n_s$). A ressource conflict occurs when I_i and I_j attempt to simultaneously use a given resource r at the same time point t_u (see figure 2). Let s_i and s_j be the related pipeline stages in which the resource r is used by the instructions I_i and I_j, respectively. Assuming a linear pipeline execution of instructions, i.e. no pipeline freeze or stall happens, the use time point t_u is equal to $(t_{oi} + s_i)$ and $(t_{oj} + s_j)$, respectively, where the variables s_i and s_j are used here as the ordinal values of some pipeline stages. Hence, the timing condition for the resource conflict is equivalent to $(t_{oj} - t_{oi}) = (s_i - s_j)$. Considering a simultaneous use of a resource r at a given clock cycle (stage level), the resource conflict occurs in reality only if this resource is used at the same phase of the clock, since a multi-phased non-overlapping clock is used (figure 3). Let p_i and p_j be the related clock phases by which the resource r is used by the instructions I_i and I_j,

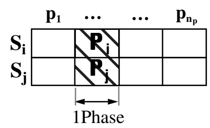

Figure 3: Phase Resource Conflict

respectively. The accurate resource conflict condition is thus equivalent to $(p_i = p_j)$.

Using the predicate *Phase_used* (I_i, s_i, p_i, r) which implies that the resource r is used by the instruction I_i at the pipeline stage s_i and the clock phase p_i, the resource conflict predicate *Resource_Conflict* is defined as follows:

$$\begin{array}{l} Resource\_Conflict\ ((I_i,\ t_{oi}),\ (I_j,\ t_{oj}),\ r) := \\ \quad \exists\, s_i,\ s_j\!: pipeline\_stage. \\ \quad\quad \exists\, p_i,\ p_j\!: clock\_phase. \\ \quad\quad\quad (0 < (t_{oj} - t_{oi})) \wedge ((t_{oi} + s_i) = (t_{oj} + s_j)) \wedge (p_i = p_j) \wedge \\ \quad\quad\quad (Phase\_used\ (I_i, s_i, p_i, r) \wedge Phase\_used\ (I_j, s_j, p_j, r) \end{array}$$

The predicate *Phase_used* is extracted from the specifications of the instruction, stage and phase levels (refer to [12]) and is computed either to true or to false.

4.2 Resource Conflict Verification

The existence of resource conflicts will degrade the performance of a RISC core, since stalls have to be used to postpone the execution of an instruction, if they exist. Our goal is to show that for all instructions and resources, the predicate *Resource_Conflict* is never true. Using the formal definition of resource conflicts, the goal is expressed formally as:

$$\begin{array}{l} \forall I_i\ I_j\!: Class\_Instruction. \\ \quad \forall t_{oi}\ t_{oj}\!: time. \\ \quad\quad \forall\ r\!: Resource. \\ \quad\quad\quad \neg\, Resource\_Conflict\ ((I_i,\ t_{oi}),\ (I_j,\ t_{oj}),\ r) \end{array}$$

This goal is hierarchically expanded at the stage and phase levels and the proof leads either to *True* or to a number of subgoals which explicitly include a specific resource and the specific stage/phase instructions which conflict. For example a conflict due to the resource *PC* between the IF stage instruction of ALU-class and the ID stage instruction of CONTROL- class is output as follows:

$$\begin{array}{l} (I_i = ALU),\ (I_j = CONTROL),\ (s_i = IF), \\ (s_j = ID),\ (p_i = \phi_2),\ (p_j = \phi_2),\ (r = PC) \end{array} \quad \vdash \quad False$$

In order to ensure that the resource *PC* is not simultaneously written by the stage instructions *IF* and *ID*, the implementation EBM has to be changed appropriately, e.g. by using an additional buffer or splitting the clock cycle into more phases.

Considering a RISC core with separate instruction and data memories (Harvard architecture [4]), the hardware implementation should ensure that no resource conflicts ever occur, i.e. formally:

$$EBM \ \vdash\ \neg\, Resource\_Conflict$$

5. Data Conflicts

Data conflicts arise when an instruction depends on the results of a previous instruction [4, 7]. The term data refers either to the content of some register within the processor or to the content of the data memory. 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) [4, 7]. Let I_i be an instruction that is sequentially issued before an instruction I_j. A RAW conflict occurs when I_j *reads* a source before I_i *writes* it, a WAR conflict happens when I_j *writes* into a destination before I_i *reads* it and a WAW conflict occurs when I_j *writes* into a destination before I_i *writes* it.

The RAW conflict is the most frequent data conflict kind. The WAR and WAW conflicts, however, are less severe and rarely occurs except in some special cases. The formal specifications and the proofs of all these data conflicts are similar, hence in the rest of in this paper we will focus on RAW data conflicts for illustration purposes.

5.1 Data Conflict Specification

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} $(t_{oi} \leq t_{ui})$. Let I_j be another instruction that is issued at a later time t_{oj} $(t_{oi} < t_{oj})$ 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. Assuming a linear pipeline execution of instructions, i.e. no pipeline freeze or stall happen, the use time points t_{ui} and t_{uj} are equal to $(t_{oi} + s_i)$ and $(t_{oj} + s_j)$, respectively. Hence, the timing condition for the RAW conflict, i.e. $(t_{uj} \leq t_{ui})$, is equivalent to $(t_{oj} - t_{oi}) \leq (s_i - s_j)$.

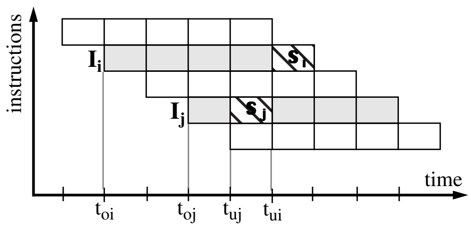

Figure 4: RAW Data Conflict

A special case of the data conflict timing condition happens when $t_{ui} = t_{uj}$, i.e. both instructions simultaneously use the resource r. In this case the conflict should be examined at the phase level. According to figure 5, a RAW data conflict at the phase level happens when the resource r is written by I_i in s_i at a clock phase p_i that occurs after clock phase p_j, where it is read by I_j in s_j. Since instructions at the phase level are executed purely in parallel, they all have the same issue time τ_o (figure 5). The additional timing condition $(\tau_{ui} \geq \tau_{uj})$ is thus equivalent to $(\tau_o + p_i \geq \tau_o + p_j) = (p_i \geq p_j)$.

Let *Range* (I_i, s_i, p_i, r) be a predicates which implies that the resource r is *written* by the instruction I_i in pipeline stage s_i at clock phase p_i and let *Domain* (I_i, s_i, p_i, r) be a similar predicate

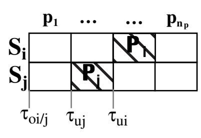

Figure 5: Phase RAW Conflict

that is true when the resource *r* is *read* by I_j in s_j at p_j. A formal specification of the RAW data conflict is thus given as follows:

$$
\begin{aligned}
& RAW\_Data\_Conflict\ ((I_i,\ t_{oi}),\ (I_j,\ t_{oj}),\ r) := \\
& \quad \exists\, s_i,\ s_j\colon pipeline\_stage. \\
& \quad\quad \exists\, p_i,\ p_j\colon clock\_phase. \\
& \quad\quad\quad (0 < (t_{oj} - t_{oi})) \wedge \\
& \quad\quad\quad [((t_{oj} - t_{oi}) < (s_i - s_j)) \vee (((t_{oj} - t_{oi}) = (s_i - s_j)) \wedge (p_i > p_j))] \wedge \\
& \quad\quad\quad Range\ (I_i,\ s_i,\ p_i,\ r) \wedge Domain\ (I_j,\ s_j,\ p_j,\ r)
\end{aligned}
$$

The *Range* and *Domain* predicates are extracted from the specification of the class level instructions at the clock cycle granularity (refer to [12]) and are computed either to false or to true, e.g. $Domain\,(ALU, ID, \phi_2, RF) = True$ (refer to table 1).

The WAR and WAW predicates are defined in a similar manner, where the semantics of the corresponding data conflict is reflected by the order of the *Range* and *Domain* predicates.

5.2 Data Conflict Verification

Our ultimate goal in the proof of the non existence of data conflicts relies in showing that none of the data conflict predicates is true. 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. Formally, the goal to prove for RAW data conflicts is given as follows:

$$
\begin{aligned}
& \forall I_i\ I_j\colon Class\_Instruction. \\
& \quad \forall t_{oi}\ t_{oj}\colon time. \\
& \quad\quad \forall\, r\colon Resource. \\
& \quad\quad\quad \neg\, RAW\_data\_Conflict\ ((I_i,\ t_{oi}),\ (I_j,\ t_{oj}),\ r)
\end{aligned}
$$

The proof adapted for this goal is constructive, i.e. if conflicts occur, the corresponding instructions, resources and the conflict timing conditions are explicitly output to the user. 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 $(s_i - s_j)$ = “*WB - ID*”:

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

This timing information gives exactly the maximum number of pipeline slots or bypassing paths that have to be provided by the software scheduling technique or the implementation EBM, respectively. In other words, the issue time of a LOAD-instruction followed by an ALU-instruction should be at least 3 time units apart. Using instruction scheduling, the needed software constraint (assumption) could then be defined as:

$$
\begin{aligned}
SW\_Constraint := ((I_i = LOAD) \wedge (I_j = ALU) \wedge \\
(r = RF) \wedge (0 < (t_{oj} - t_{oi}))) \Rightarrow ((t_{oj} - t_{oi}) > 3)
\end{aligned}
$$

Another widely used data conflict resolution technique is *bypassing* (also called *forwarding*) [4]. A bypassing technique ensures that the needed data is forwarded as soon as it is computed (end of the EX-stage) to the next instruction (begin of the EX-stage). This is implemented by using some registers and feedback paths that hold and forward these data, respectively. Formally, the existence of such registers and corresponding forward paths can be specified as follows:

$$
\begin{aligned}
Bypassing := \exists rb\colon Resource\,.\,(rb = RF) \wedge \\
Range\,(I_i, EX, p_i, rb) \wedge Domain\,(I_j, EX, p_j, rb)
\end{aligned}
$$

Having this bypassing condition, the existentially quantified pipeline stage variables s_i and s_j in the definition of RAW are set to EX and the timing condition is then reduced to:

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

which is always true.

Having implemented the bypass technique as part of the EBM and/or assuming the mentioned software constraints, we are able to prove the non existence of data conflicts:

$$
SW\_Constraints,\ EBM \quad \vdash \quad \neg\, Data\_Conflict
$$

6. Control Conflicts

Control conflicts arise from the pipelining of branches, jumps, traps and other instructions that change the program counter *PC* [4]. The presence of such control instructions in the instruction stream may lead to an interruption of the linear instruction flow.

In highly pipelined processors, the next instruction fetch may begin long before the current instruction has been fully decoded and executed. Thus it may be impossible to correctly update the machine’s program counter *PC* before the next few instructions are fetched. If one instruction is issued per clock, and a jump instruction takes *N* cycles to fetch and execute, then the *N-1* instructions following the jump will always be executed, since they will have been fetched before the program counter *PC* was updated. Thus straightforward program coding may yield incorrect results.

It can be seen from the discussion above that a control conflict usually occurs when an instruction attempts to read the resource *PC* that is not yet updated (written) by a previous instruction. Hence, the control conflict definition complies with the definition of a RAW data conflict in *PC* [7] and the control conflict predicate could be defined as follows:

$$
\begin{aligned}
& Control\_Conflict := \\
& \quad RAW\_Data\_Conflict\ ((CONTROL,\ t_{oi}),\ (I_j, t_{oj}),\ PC)
\end{aligned}
$$

The conflict freedom proof is therefore only a special case of the data conflict proofs and is totally covered by it. For conflict resolution no bypassing is possible, since the calculation of the target address cannot be done earlier. Using the software scheduling technique for the DLX processor, we just need one delay slot to ensure that control instructions are executed correctly. The software constraint needed in this case is defined as follows:

$$
\begin{aligned}
SW\_Constraint := ((I_i = CONTROL) \wedge (r = PC)) \\
\Rightarrow ((t_{o(i+1)} - t_{oi}) > 1)
\end{aligned}
$$

Using these appropriate software constraints, the non existence of control conflicts is formally ensured, i.e.:

$$
SW\_Constraints \quad \vdash \quad \neg\, Control\_Conflict
$$

7. Experimental Results

For the validation of our whole methodology, we have used the DLX RISC processor as a realistic benchmark. The DLX processor is a 5 stage pipeline RISC processor with a 2 phased clock. The DLX 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 [10]. We have implemented the hardware for the DLX core in a commercial VLSI design environment using a 1.0 μm 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.

All formal specifications and proofs of our methodology are implemented within the HOL verification system [3]. The implementation of the interpreter model specifications and the proof of step (2) is reported in [12]. The overall specification text is about 4760 lines long and this part of the proof for the whole DLX processor core took about 457 secs on a SPARC10 with a 96 MB main memory.

For the proof of step (1), we have implemented for each kind of conflict general specifications and proof strategies. Further, all implemented specifications and proof scripts are parameterized and kept generic, so that the implementation could be used for a wide range of other RISC processors. The overall specification and proof script text of step (1) is about 1780 lines. The run times for the pipeline correctness proof of the DLX processor on a SPARC10 with a 96 MB main memory are given in table 2. The overall pipeline correctness proof of the DLX, including the generation of the *Phase_used*, *Range* and *Domain* predicates, is therefore about 47 min. and 33 sec.

Table 2: Formal Proof Results

Verification Goal	Time (sec.)	Comments
Phase_used Gen.	247.26	–
Range & Domain Gen.	221.06	–
Resource Conflicts	457.66	0 conflicts
RAW Data Conflicts	655.67	9 conflict cases in RF (3 slots) and 4 conflicts in PC (1 slot)
WAR Data Conflicts	551.77	0 conflicts
WAW Data Conflicts	680.86	0 conflicts
Control Conflicts	39.25	4 conflict cases (1 slot)
Σ Pipeline Correctness	2853.53	–

8. Conclusions and Future Work

We have shown that pipeline conflicts which occur in RISC cores – resource conflicts, data conflicts and control conflicts can be conveniently modelled at various abstraction levels using higher-order predicates. The use of the hierarchical RISC interpreter model and in particular the exploitation of the class level, allows us to derive compact specifications of the conflicts that can occur in the pipeline. We have implemented constructive proofs for these conflicts and hence the designer gets invaluable feedback for introducing conflict resolution mechanisms; either by making appropriate modifications to the hardware or by generating the required software constraints.

The model, the specification and the proof techniques are generic in the sense that it is applicable to any RISC core. Since the RISC interpreter model closely reflects the RISC design hierarchy, the specifications in higher-order logic are easy to derive. Given such specifications and an implementation, the proof process has been automated by using parametrizable proof scripts. These proof scripts are independent of the underlying implementation and can be used for a large number of RISC cores. The entire methodology has been validated by using the DLX processor.

The run times of the proofs shown in the table 2, illustrate the feasibility of formal verification techniques when applied intelligently to specific classes of circuits. 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] 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.

[3] Gordon, M.; Melham, T.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic; Cambridge, University Press, 1993.

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

[5] Hunt, W.: Microprocessor Design Verification; Journal of Automated Reasoning, Vol. 5, 1989, pp. 429-460.

[6] Joyce, J.: Multi-Level Verification of Microprocessor-Based Systems; Ph.D. Thesis, Computer Laboratory, Cambridge University, December 1989.

[7] Kogge, P.: The Architecture of Pipelined Computers; McGraw-Hill, 1981.

[8] 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.

[9] Srivas, M.; Bickford, M.: Formal Verification of a Pipelined Microprocessor; IEEE Software, September 1990, pp. 52-64.

[10] Tahar, S.; Kumar, R.: A Formalization of a Hierarchical Model for RISC Processors; In: Spies, P. (Ed.), Proc. European Informatics Congress Computing Systems Architecture (Euro-ARCH93), Munich, October 1993, Informatik Aktuell, Springer Verlag, pp. 591-602.

[11] Tahar, S.; Kumar, R.: Towards a Methodology for the Formal Hierarchical Verification of RISC Processors; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, pp. 58-62.

[12] Tahar, S.; Kumar, R.: Implementing a Methodology for Formally Verifying RISC Processors in HOL; Proc. International Meeting on Higher Order Logic Theorem Proving and its Applications (HUG93), Vancouver, Canada, August 1993, pp. 283-296.

[13] Windley, P.: The Formal Verification of Generic Interpreters; Ph.D. Thesis, Division of Computer Science, University of California, Davis, July 1990.