Technical Report No. FZI 9/95
Forschungszentrum Informatik
Karlsruhe, Germany, August 1995

A Practical Methodology for the Formal Verification of RISC Processors

SOFIÈNE TAHAR tahar@iro.umontreal.ca
IRO Department, University of Montreal, Montréal (Québec), H3C 3J7 Canada

RAMAYYA KUMAR kumar@fzi.de
FZI, Haid-und-Neu Straße 10-14, 76131 Karlsruhe, Germany

Abstract. In this paper a practical methodology for formally verifying RISC cores is presented. This methodology is based on a hierarchical model of interpreters which reflects the abstraction levels used by a designer in the implementation of RISC cores, namely the architecture level, the pipeline stage level, the clock phase level and the hardware implementation. 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 independent steps. The first step shows that each architectural instruction is implemented correctly by the sequential execution of its pipeline stages. The second step shows that the instructions are correctly processed by the pipeline in that we prove that under certain constraints from the actual architecture, no conflicts can occur between the simultaneously executed instructions. This proof is constructive, since the conditions under which the conflicts occur are explicitly stated thus aiding the user in its removal. All developed specifications and proof scripts are kept general, so that the methodology could be used for a wide range of RISC cores. In this paper, the described formalization and proof strategies are illustrated via the DLX RISC processor.

Keywords: Formal Specification, Hardware Verification, Higher-Order Logic, RISC Processors, Processor Verification, Pipeline Verification

1. Introduction

As computer systems are becoming increasingly complex, the trustworthiness of their design is questionable. Conventional approaches such as simulation and testing have a very high cost to confidence-gain ratio and furthermore, the correctness of the design cannot be guaranteed due to the combinatorial explosion of test vectors [20]. This situation is particularly unsatisfactory in the case of embedded computers for safety-critical systems, such as aircraft, spacecraft and nuclear reactor control etc., where design errors could lead to loss of life and expensive property [74]. Hence, there is a need to produce high-integrity processors that are correct in *all* situations. Although completely reliable systems cannot be guaranteed, the use of *formal methods* [39] is an alternative approach that systematically analyses *all* cases in a design and specification [23].

In the recent past several successful microprocessor specification and verification efforts have been performed using formal methods; some using high-order logic [22, 37, 38, 49, 77] and others based on functional calculi [8, 25, 44, 64, 66]. Among the processors verified within these works only the VIPER [22] and the C/30 [25] processors are commercial ones, however, their verification was only partly achieved. With exception of these two processors, all related works on microprocessor verification deal with very simplified processors, so-called toy machines, when compared with today's commercially available microprocessors. Furthermore, except the work of Windley [77], these efforts were concerned with a specific microprocessor and do not give any general methodology.

During the verification of processors, powerful specifications are needed to express the functionality, temporal aspects and structure at different levels of abstraction. Due to the expressiveness of higher-order logic in specifying complex circuits at different abstraction levels, the formalism used in our work will be based on this powerful logic. But, since this logic is neither complete nor decidable [4], no automated proofs will be provided in general. However, this disadvantage can be circumvented by the development of appropriate heuristics and techniques which automate the verification of a special class of circuits, e.g. microprocessors, arithmetic circuits, protocol circuits, systolic arrays, signal processors, etc. This is due to the observation that specific classes of circuits have very typical syntactic structures which can be exploited to provide automation.

Microprocessors build a particular class of hierarchical circuits, that are increasingly used in a wide range of applications. A look at the microprocessor market shows that there are two kinds of design philosophies: CISCs (Complex Instruction Set Computers) and RISCs (Reduced Instruction Set Computers). Most related work in microprocessor specification and verification were concerned with microprogrammed non-pipelined processors [37, 44, 49, 77]. Although large examples have been verified [38, 45] and a general methodology for verifying microprogrammed processors has been given [77], these efforts do not reflect the complexity of the commercially available CISC processors. Our studies of real CISC microprocessors have shown that they have a very unstructured and dirty design including a large control part (approx. 70% of the chip area) encoded in an intricate way [75]. This complexity is the reason why conventional validation methods such as logic simulation or breadboarding are the major bottleneck in CISC microprocessor design projects [75]. Therefore, no reasonable methodology can be set up for the verification of commercial CISC processors.

The RISC philosophy is based on the idea of pushing the complexity from the hardware to the software. This characteristic leads to a much simpler design with a higher throughput. In contrast to CISCs, RISC designs are better structured and hence more tractable for using formal methods [12]. However, additional problems such as pipelining have to be tackled since they form the essence of RISCs. Moreover, contemporary RISCs include complex features, such as floating point operations, memory management, etc. that have to be considered within the verification process. However, due to the regularity of a RISC design and the use of modular implementations, the overall architecture of such processors can be defined using a multiple layered architecture [7], consisting of the core architecture, the numerical architecture and the protected architecture (figure 1). The *core architecture* executes the basic instruction set of the RISC processor, includes the basic instruction pipeline and controls the whole microprocessor. The *numerical architecture* provides support for floating point and complex arithmetic operations. The *protected architecture* is for memory management, multitasking and multiprocessing tasks. The more one moves from the innermost ring (RISC core) to the outermost one, the more are the differences in architectures from one RISC implementation to another, e.g. use of different cache mechanisms. Since the objective of our endeavour is to provide a *general* methodology, we will therefore concentrate on the core architecture of a RISC processor, as a first step towards the verification of whole RISC processors. The handling of upper layers is topic of future work and is not covered by the scope of this paper.

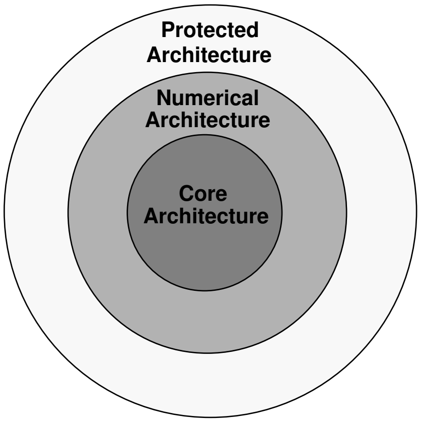

Figure 1. Multiple Layered RISC Architecture

Recently, there have been successful efforts for verifying pipelined processors using theorem-provers [1, 15, 26, 63, 65, 66]. However, in all these cases, either the processor was extremely simple (e.g. in [26, 65] a very simple 3-stage pipeline known as Saxe-pipeline is handled) or a large amount of labor was required. Among these works, only the work in [15] deals with the verification of a RISC processor, namely a SPARC model [68]. Still, this work was only able to verify parts of the processor at certain levels of abstraction. Lately, automated techniques for the verification of pipelined processors have been presented [9, 16]. However, due to the computational cost of BDD manipulations [13], the method presented in [9] was only able to prove the correctness of simplified pipelined processor examples (e.g. using one single general purpose register, few 4-bit ALU-operations, etc.) and that in [16] deals with the verification of the control part and additionally, abstracts the behaviour of the datapath components. Moreover, these works do not reflect the overall behaviour of real RISC processors (e.g. only few instructions, no interrupts, etc.). Besides these works on pipelined processors, there exist only few publications on the formal verification of pipelined hardware circuits in general which however do not address the problems of RISC pipelines [11, 14, 36]. In contrast to all related works, we are developing a methodology and an associated environment for the routine verification of RISC cores in their entireties, i.e. from the specification of instruction sets down to their circuit implementations, independent of the data width and including features of real RISCs as bypassing, delayed execution, interrupts, etc.

With the aim of advancing the state of technology in hardware verification, we set up the following goals:

- To develop a methodology for the verification of a particular class of circuits, i. e. RISC cores
- To formally reason about new aspects in microprocessor design which were not sufficiently addressed by previous efforts (especially pipelining)
- To set up advanced techniques for the verification of real RISCs, that are not designed just for the purpose of verification
- To elaborate practical tools that automate the verification process using higher-order logic in a theorem prover environment
- To use this methodology as a framework for formally specifying and verifying a broad range of large, realistic RISC cores
- To implement this framework in the HOL theorem prover [34] and to integrate it into a general verification framework MEPHISTO [52]

The organization of this paper is as follows: Section 2 describes a novel hierarchical model for RISCs on which the verification process will be based. Section 3 first sketches a new temporal abstraction mechanism and then gives a formalization of the specification of this model. Section 4 describes the management of the verification tasks which will be explored in detail in the following sections 5 and 6. Section 7 briefly describes some aspects of the implementation of the presented methodology in HOL. Section 8 contains some experimental results based on the verification of a VLSI implemented RISC processor and section 9 finally concludes the paper. It is to be noted, that for illustration purposes, most of the methods and techniques presented in this paper are being exercised by means of a RISC example — DLX [43]. This processor is an hypothetical RISC which includes the most common features of existing RISCs such as Intel i860, Motorola M88000, Sun SPARC or MIPS R3000.

2. RISC Verification Model

Some recent work has shown that the specification and verification of microprogrammed processors can be simplified through the insertion of intermediate abstraction levels, called *interpreters*, between the specification as an instruction set and the hardware implementation [45, 49, 77]. The overall approach of interpreters used reflects the way microprogrammed microprocessor designs are carried out and designed [3]. Each interpreter consists of a set of visible states and a set of state transition functions which define the semantics of the interpreter at that level of abstraction. At the architecture level, for example, states such as the program counter, register file or data memory, etc. are visible and the set of transition functions corresponds to the instruction set of the processor. 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. Using this interpreter model, it is sufficient to prove that each level correctly implements the next abstraction level instead of verifying that each instruction is correctly implemented by the hardware. 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.

2.1. CISC Interpreter Model

In some related work an interpreter model for microprogrammed processors has been presented [5, 49, 77]. This CISC interpreter model is given in figure 2 (where the arrow between the levels means that the upper level specification is an abstraction of the next lower one). It comprises the macro, micro 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 (Electronic Block Model). The macro level reflects the programmer's view of instruction execution. At the micro level, an instruction is interpreted by executing a sequence of microinstructions. The phase level description decomposes the interpretation of a single microinstruction into the execution of a set of elementary operations. Using this interpreter model the verification task is replaced by several simplified proof steps. For example, one has only to prove that the EBM implements 4 to 6 phases instead of directly implying the whole instruction set. However, as mentioned earlier, this model has been applied for very simplified microprocessors and is not usable for complex real CISC processors.

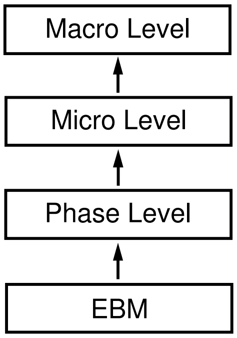

Figure 2. CISC Interpreter Model

The way RISC designs are carried out and structured is different from that of CISCs, e.g. because of the hardwired control the micro level does not exist more. The mentioned structuring of the specification using the CISC interpreter model is hence unsuitable for RISC cores [70] and we have to look for another verification model.

2.2. *A Novel RISC Interpreter Model*

A RISC processor executes each instruction in a number of physical steps, called *pipeline stages* (e.g. IF, ID, EX, WB, for instruction fetch, instruction decode, instruction execution and result write back, respectively). The duration of a pipeline stage corresponds to one machine clock period. We define a *stage instruction* as the set of transfers, which occur during the corresponding pipeline segment. Using a multiple phase non-overlapping clock, each stage operation is partitioned into a number of clock phase operations. We define a *phase instruction* of a specific stage as the set of the parts of the transfers that occur during that clock phase.

The instruction set of a RISC core is simple, elementary and less encoded, thus the complexity of RISC instructions can be compared to that of CISC microinstructions [35]. The pipeline stages are also comparable to CISC phases, since their number is limited by the pipeline depth, and is constant for almost all instructions. The RISC phases could be compared to CISC instances, which are refinements of clock phase operations at the asynchronous level [3]. Using this analogy, a naive model for RISCs, similar to that of CISCs, could be given. This model is (top-down) built up of an architecture level, a stage level, a phase level and an implementation EBM. However, in contrast to CISCs, the RISC phase instructions are stage dependent and the stage instructions differ from one instruction to another. Using such a model, the number of phase instructions and therefore the number of verification steps between the EBM and the phase level is $N_a * n_s * n_p$, where N_a, n_s and n_p are the number of architectural instructions, pipeline stages and clock phases, respectively. Since the complexity of the proof between the EBM and the next abstraction level is the largest [22], the use of a naive interpreter model does not yield any advantages, e.g. with $N_a = 80$, $n_s = 5$ and $n_p = 4$, a naive calculation would have yielded $80 * 5 * 4 = 1600$ different phase instructions that have to be specified and verified resulting into 1600 single theorems.

As a first solution to reduce this number, we exploit the notion of instruction classes[1] [70]. An *instruction class* intuitively corresponds to the set of instructions with similar semantics, e.g. ALU, FLP, LOAD, CONTROL for arithmetic-logic, floating point, load and control instructions, respectively. Generally, instruction classes are implicitly provided by the instruction set of each

---

[1] This Notion of instruction classes is being also adapted by RISC designers in order to improve the pipeline execution [43] as well as for simulation [76], synthesis [21] or testing purposes [69].

RISC processor [29, 48, 58, 68]. Furthermore, a group of instructions belonging to one class usually use the same number of pipeline stages, are executed by the same stage instructions and are usually implemented in hardware by the same type of functional unit. For example, all binary arithmetic and logic operations (+, –, $\wedge$, $\vee$, etc.) can be abstracted by a single operator called “*op*”. The stage and phase instructions can now be parameterized in accordance to the class abstraction, i.e. they are not dependent on each architectural instruction but only on the instruction class. Thus the total number of different stage and phase level instructions can be reduced to $N_s = N_c * n_s$ (N_c is the number of classes) and $N_p = N_c * n_s * n_p$, respectively. The class level is therefore introduced as the top level of our interpreter model.

Real RISC cores show further regularities which can be incorporated into our interpreter model. A closer look at the stage level shows that some stage instructions are common to more than one class (e.g. in general the IF-stage instruction is shared by all classes), and additionally some classes do not require the full pipeline depth, e.g. in order to accelerate the execution of control instructions (e.g. jumps, branches, etc.) only 2 pipeline stages are used. This implies that the number of different possible stage instructions N_s is much less then $N_c * n_s$. 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 this observation into our model yields N_p (the number of different potential phase instructions) as $N_p \ll N_s * n_p$.

Table 1. DLX Pipeline Structure.

	ALU	**LOAD**	**STORE**	**CONTROL**
IF	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4	IR ← I-MEM [PC] PC ← PC+4
ID	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 ← IR	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 ← IR	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 ← IR	BTA $\underset{\phi_1}{\leftarrow}$ $\mathbf{f_C}$ (PC, IR, RF) PC $\underset{\phi_2}{\leftarrow}$ BTA
EX	ALUOUT ← A **op** B	DMAR ← A+(IR1)	DMAR← A+(IR1) SMDR ← B	
MEM	ALUOUT1 ← ALUOUT	LMDR ← $\mathbf{f_L}$(D-MEM[DMAR])	D-MEM[DMAR]← $\mathbf{f_S}$(SMDR)	
WB	RF[rd] $\underset{\phi_1}{\leftarrow}$ ALUOUT1	RF[rd] $\underset{\phi_1}{\leftarrow}$ LMDR		

Table 1 shows the pipeline structure of DLX [43] which has four instruction classes ALU, LOAD, STORE and CONTROL, five pipeline stages IF, ID, EX, MEM, and WB and two clock phases ϕ_1 and ϕ_2. This pipeline structure lists the set of transfers which occur at the stage and phase levels of the DLX processor, where the rows and columns represent the pipeline stages and the instruction classes, respectively. In table 1, “←” represents that the stage transfer is not broken down into phase transfers. “$\underset{\phi_1}{\leftarrow}$” , “$\underset{\phi_2}{\leftarrow}$” represent that the transfers take place in phase 1, 2, respectively. Further, the class abstractions are reflected through the class abstraction functions *op*,

f_L, f_S and f_C that are used in related pipeline stages[2]. In the rest of the paper we will denote stage and phase instructions as follows:

- stage instructions: IF_A, ID_C, EX_S, MEM_L, etc.
- phase instruction: $\phi_1 IF_A$, $\phi_2 ID_C$, $\phi_1 EX_S$, $\phi_2 MEM_L$, etc.

where the subscripts of the pipeline stage identifiers represent the first letter of the corresponding instruction class. For example, EX_S means the EX-stage instruction of the STORE-class which is composed of the stage operations "$DMAR \leftarrow A + (IR1)$" and "$SMDR \leftarrow B$" in table 1.

Using this DLX architecture the number of class, stage and (potential) phase instructions is $N_c = 4$, $N_s = 11$ and $N_p = 16$, respectively.

The overall hierarchical model obtained is given in figure 3. The architecture, class, stage and phase levels correspond to the set of architectural, class, stage and phase instructions, respectively. Each of these levels corresponds to a refinement of the processor behaviour and could be specified independently. Summarizing the overall specification of the different abstraction levels, we have to specify:

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

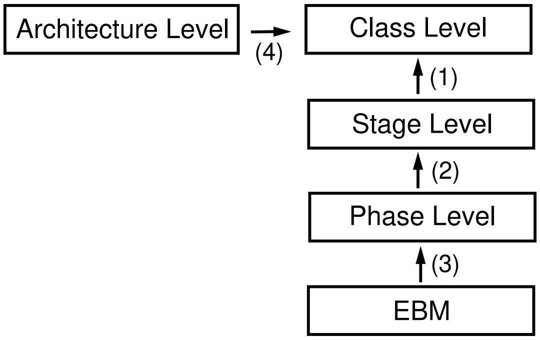

Figure 3. RISC Interpreter Model

Using the steps (1), (2) and (3), as shown in the above figure, 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. Moreover, each verification step can be done independently and should help the designer in successively refining and verifying the design. The verification steps (1) and (2) are expected to be relatively straightforward while step (3) seems to be the hardest one, since the EBM is a complex structural description and the phase instructions are behavioural [23]. However, the proofs are quite similar in nature and a strategy can therefore be evolved.

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 of

---

[2] Instead of the infix operator *op* a class abstraction function in prefix form f_A could be used in an equivalent manner, i.e. $A \; op \; B \equiv f_A (A, B)$.

the actual architecture by replacing the class abstraction function by concrete operations. Additionally, we should also prove the statement that the instruction classes abstract all instructions of the architecture (step (4) in figure 3).

Using the above presented hierarchical structuring of the verification process, the proofs can be managed hierarchically in a top-down or bottom-up manner, so that a *verification-driven design* or a *post-design verification* can be performed. By a top-down verification-driven design, we mean that the verification and design process are interleaved, so that the verification of a current design status, against the specification, yields the necessary constraints for the future design steps. A post-design verification is verification in the normal sense which is performed in a top-down or bottom-up manner after the entire design is completed at all levels. Within the scope of this paper, the correctness proofs will be mainly handled by means of the top-down verification-driven design methodology.

3. Formal Specification

Within this verification model, we have different abstraction levels which have to be related to each other. There are four kinds of abstractions: structural abstraction, behavioural abstraction, data abstraction and temporal abstraction [55]. Before getting into the details of the formal specifications of the model levels, we will briefly discuss the structural, behavioural and data abstractions and then we will focus on the concept of temporal abstraction in a dedicated subsection.

Structural and behavioural abstractions are natural consequences of the hierarchical model. In our RISC model, the structural abstraction is reflected by the visible state components at each hierarchy level. Starting from the hardware implementation EBM, it includes all state components of the machine. Since the phase level is only a behavioural abstraction of the EBM, all state components of the EBM are visible at this level too. The stage and class levels, however, abstract the structures of the lower levels and include subsets of the visible state components of the phase and stage levels, respectively. Furthermore, since the class level is only a behavioural abstraction of the architecture level, they use the same structural components, i.e. the programming model. Regarding data abstraction, throughout our approach, we will let the specifications be based on the data types that the microprocessor is to manipulate, i.e. bit-vectors. In [49, 77] uninterpreted data types were used. Here we use concrete data types of bit-vectors, e.g. from the bit-vector library of HOL [79]. Since bit-vectors are naturally used in the description of the instruction set, we do not make any data abstractions and use bit-vectors through all abstraction levels. Thus, we save a lot of mapping functions between the concrete data type and an abstract one, e.g. natural numbers.

3.1. Temporal Abstraction

Temporal abstraction relates the different time granularities which occur in the formal specifications at various levels of abstractions [55]. The class and the instruction levels use the same time granularity, which corresponds to instruction cycles. The stage level granularity is that of clock cycles and the phase level granularity corresponds to the duration of single phases of the clock (figure 4).

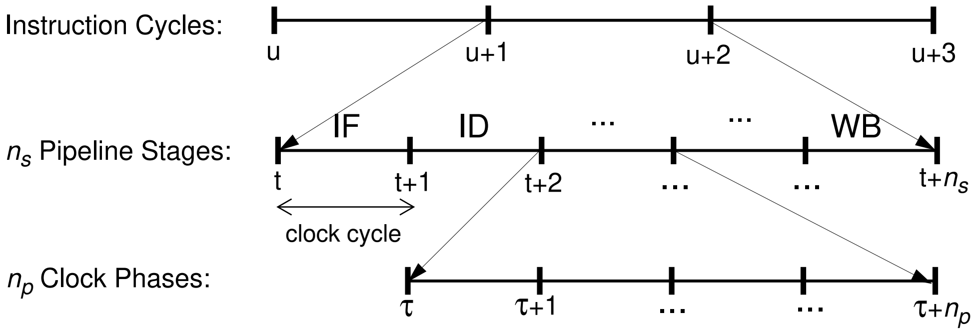

Figure 4. Time Granularities of the RISC Model

Temporal abstraction allows us to hide the unnecessary details at higher levels of abstractions [55]. A fundamental step in the formalization of the interpreter model will be to establish a mathematical relationship between the abstract time scales and the next concrete ones. In a sequential machine the effects of one instruction can be considered at the end of an instruction cycle, since they are caused by this specific instruction (figure 5). On the other hand, in a pipelined machine (as is the case for RISC processors), instructions are executed in an overlapped manner (figure 5), where state changes of the machine during one instruction cycle cannot be related to only one instruction. Therefore the abstract discrete time unit of RISC instruction cycle cannot be directly related to a fixed discrete time point on the concrete clock time scale.

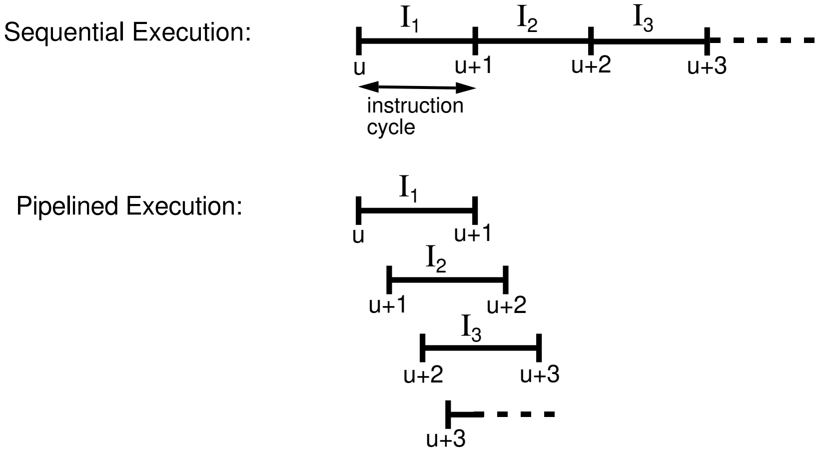

Figure 5. Sequential and Pipelined Executions

Referring to the instruction set manual of a specific processor, the semantics of an instruction is given as a state transition occurring in an instruction cycle with an implicit time relationship. For example, the semantics of an ADD instruction is defined as follows:

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

where *RF* is the register file and *rd*, *rs1*, *rs2* are the destination and source addresses of some registers in the register file. These addresses correspond to fields of the actual instruction word, which is addressed by the program counter *PC*.

Using the abstract time of instruction cycles (represented by the variable "*u*"), this instruction can be described formally by means of a predicate involving time as follows, where *I-MEM* and *D-MEM* represent the instruction and data memory, respectively[3]:

$$ADD\_SPEC\ (PC, I\text{-}MEM, RF, D\text{-}MEM) :=$$
$$\forall\ \mathbf{u}: Inst\_cycle.\ RF(\mathbf{u+1})[rd(\mathbf{u})] = RF(\mathbf{u})[rs1(\mathbf{u})] + RF(\mathbf{u})[rs2(\mathbf{u})] \qquad \text{(i)}$$

Referring to table 1, the ADD-instruction, using the more concrete time granularity of clock cycles (represented by the variable "*t*"), can be specified formally as:

$$ADD\_IMP\ (PC, I\text{-}MEM, RF, D\text{-}MEM) :=$$
$$\forall\ \mathbf{t}: Clock\_cycle.\ RF\ (\mathbf{t+5})[rd(\mathbf{t})] = RF(\mathbf{t+1})[rs1(\mathbf{t})] + RF(\mathbf{t+1})[rs2(\mathbf{t})] \qquad \text{(ii)}$$

A mapping function that relates abstract time scales in (i) to concrete ones in (ii) is not linear since an unit of time on the abstract time scale does not necessarily correspond to one discrete time point of the next concrete time scale. For example, the same time point u, that is used in (i) for computing the addresses $rs1$, $rs2$ and rd, as well as for reading the register file *RF*, has to be related to t and $t+1$, corresponding to the IF and ID-stage, respectively.

In a pipelined execution, state changes at the abstract level can take place at some time between the two discrete end-points of its time interval, i.e. at some time between u and $u+1$ depending upon the implementation. A mapping time abstraction function should therefore take some implementational contexts into consideration while converting the abstract time to a more concrete one [70]. A context parameter could be given as a tuple involving a read/write information and a pipeline stage identifier. For example, [*read, ID*] and [*write*, *WB*] describe a read operation during the ID-stage and a write operation during the WB-stage, respectively. Let f_t be this temporal abstraction function, applying f_t to the instruction cycle time variable u using the above context examples, we obtain —"f_t ([*read, ID*], u) = $t+1$" and "f_t ([*write*, *WB*], $u+1$) = $t+5$", respectively.

At a lower level, i.e. between the stage and the phase levels, we have a similar temporal relationship since state transitions can occur at some time points within the clock cycle interval corresponding to some specific phases. In a similar manner, a time abstraction function has to be provided which takes into account an implementation dependent context parameter. The corresponding context variable is defined as a tuple composed of a read/write information and a clock phase identifier, e.g. [*read*, ϕ_1] means a read during phase 1.

Due to the similarities of the temporal behaviours at both levels, we have developed one general parameterized time abstraction function (represented by *Time_abs*) for both abstraction levels. This temporal abstraction function takes as parameters: a natural number n corresponding to the total number of implemented pipeline stages or clock phases, a context tuple C comprising the read/write information and the pipeline stage or clock phase identifier, and a time variable x. Furthermore, assuming that stage and phase identifiers are ordered in some manner (e.g. *IF* = 0, *ID* = 1, etc.), we define a function *ORD* which computes the ordinal values of the corresponding stage or phase identifiers and also the ordinal values of *read/write*, e.g. *ORD (EX)* = 2 and *ORD (write)* = 1. Additionally, this definition of the time abstraction function should take into

---

[3] The notation "x: σ" means that the variable x is of type σ.

account that written values are considered at the end of a discrete time interval while read values are those at the beginning. A possible implementation of *Time_abs* can be defined formally as follows[4]:

$$\boxed{\begin{array}{l} \vdash_{def} Time\_abs\ n\ \ C\ \ x := \\ \qquad \text{let } (rw = fst(C) \wedge Id = snd(C)) \text{ in} \\ \qquad\qquad \left(\frac{1}{n} * (x - ORD\ (rw)) + ORD\ (Id) + ORD\ (rw)\right) \end{array}}$$

This abstraction function has the advantage, that it allows specifications to be abstract and implementation independent. Moreover, due to the given parametrization, it can be used both for the verification of the class level and for the stage level. The various instantiations for n and C are given later when constructing the appropriate verification goal, where n is instantiated once according to the current abstraction level and C is set for each state component of the abstract specification (see section 5). For example, with $n = n_s = 5$, $C = [read, ID]$, $ORD(read) = 0$ and $ORD(ID) = 1$, we obtain $Time\_abs\ (n_s, [read, ID],\ u) = \frac{1}{5} * (u - 0) + 1 + 0 = \frac{1}{5}u + 1 = t + 1$. The use of the temporal abstraction function *Time_abs* will be illustrated in section 5 while describing the verification process.

3.2. Specification of the Model Levels

Each interpreter level is defined by means of a set of instructions which reflects the semantics at each level of abstraction. Each instruction can be formally specified in higher-order logic by means of a predicate whose parameters correspond to the visible states at that level of abstraction and eventually to a class abstraction function. In contrast to the architecture, class, stage and phase levels, the EBM cannot be characterized as an interpreter since it corresponds to the hardware structure. Formally, it will be described as a hierarchy of predicates specifying the different hardware components. In following, we describe the formal specification of each level of our model in a dedicated subsection. The different formalization techniques used will be illustrated by some simple examples based on the DLX processor.

3.2.1. Architecture Level

At the architecture level, the instruction set is conventionally specified as state transitions occurring in an instruction cycle which implicitly involves time. For example the semantics of an ADD and a branch-on-zero (BRZ) instruction are defined as follows:

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

$$\begin{array}{ll} BRZ := \ if\ (RF\ [rs1] = 0) & then\ \ PC \leftarrow PC + offset16 \\ & else\ \ PC \leftarrow PC + 4 \end{array}$$

where *offset16* is a 16-Bit value corresponding to a field of the instruction word addressed by the program counter *PC*.

Formally, each architectural instruction can be specified by means of a predicate using the abstract time of instruction cycles. Given that u is an unit for an instruction cycle, the above ADD and BRZ instruction examples can be described formally as follows[5]:

---

[4] *fst* returns the first component of a tuple and *snd* returns its second component.

[5] The expression “$a \longrightarrow\ b \mid c$” is an abbreviation for “if a then b else c”.

$$\begin{array}{l} \vdash_{def} ADD\_SPEC\ (PC, RF, I\text{-}MEM, D\text{-}MEM) := \\ \quad \forall\ u: Instr\_cycle. \\ \quad\quad \text{let } (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge rs2 = [I\text{-}MEM(PC)]_{20..16} \wedge rd = [I\text{-}MEM(PC)]_{15..11}) \text{ in} \\ \quad\quad\quad RF(u+1)\ [rd(u)] = RF(u)\ [rs1(u)] + RF(u)\ [rs2(u)] \end{array}$$

$$\begin{array}{l} \vdash_{def} BRZ\_SPEC\ (PC, RF, I\text{-}MEM, D\text{-}MEM) := \\ \quad\quad \forall\ u: Instr\_cycle. \\ \quad\quad\quad\quad let\ (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge offset16 = [I\text{-}MEM(PC)]_{15..0})\ in \\ \quad\quad\quad\quad\quad PC(u+1) = (RF(u)\ [rs1(u)] = 0) \rightarrow PC(u) + offset16(u)\ \vert \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad PC(u) + 4 \end{array}$$

3.2.2. Class Level

A class instruction abstracts the semantics of a group of architectural instructions. Similar to the instruction set, the semantics of class instructions can be given as state transitions. For example, the ALU class instruction (table 1) is defined as follows, where *op* abstracts all required arithmetic-logic operations:

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

Analogously, using the class abstraction function f_C (which involves all implemented control functions as jumps, branches, etc.), the CONTROL-class instruction is defined as the following state transition:

$$CONTROL := PC \leftarrow \boldsymbol{f}_{\mathbf{C}}\ (PC, offset16, offset26, RF[rs1])$$

where *offset26* is a 26-Bit value corresponding to a field of the actual instruction word. Using the parameters *PC*, *offset16*, *offset26* and *RF*[*rs1*], the function f_C can compute all required target address variants, e.g. for a register indirect jump f_C computes $PC+RF[rs1]$.

The formal specification of the class level is almost the same as that of the architecture level (since the same state components and the same time granularity are used), except that a generalized class parameter (which will be used instead of a specific operator or function) is introduced as part of the predicate parameters. Let *u* be an unit of time for an instruction cycle, the semantics for the ALU and CONTROL class instructions can be specified formally by the following predicates:

$$\begin{array}{l} \vdash_{def} ALU\_SPEC\ \ op\ (PC, I\text{-}MEM, RF, D\text{-}MEM) := \\ \quad \forall\ u: Instr\_cycle. \\ \quad\quad \text{let } (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge rs2 = [I\text{-}MEM(PC)]_{20..16} \wedge rd = [I\text{-}MEM(PC)]_{15..11}) \text{ in} \\ \quad\quad\quad\quad RF(u+1)\ [rd(u)] = RF(u)\ [rs1(u)]\ op\ RF(u)\ [rs2(u)] \end{array}$$

$$\begin{array}{l} \vdash_{def} CONTROL\_SPEC\ \ f_{\mathrm{C}}\ (PC, I\text{-}MEM, RF, D\text{-}MEM) := \\ \quad\quad \forall\ u: Instr\_cycle. \\ \quad\quad\quad \text{let } (rs1 = [I\text{-}MEM(PC)]_{25..21} \wedge \\ \quad\quad\quad\quad offset16 = [I\text{-}MEM(PC)]_{15..0} \wedge offset26 = [I\text{-}MEM(PC)]_{25..0}) \text{ in} \\ \quad\quad\quad\quad\quad PC(u+1) = f_{\mathrm{C}}\ (PC(u), offset16(u), offset26(u), RF(u)\ [rs1(u)]) \end{array}$$

3.2.3. Stage Level

A stage instruction is defined as a set of elementary state transitions, that implement the corresponding semantics. The additional visible states at the stage level are mostly pipeline buffer latches as shown in table 1, for the DLX example. Formally, a stage instruction is specified as a predicate on the visible states at this level. It is a conjunction of simple transfers that can be directly read-off from the pipeline architecture (table 1) and encoded formally. For example, the ID-stage instructions ID_A of the ALU class (see row ID and column ALU in table 1) is specified by the following predicate:

$$\begin{array}{l} \vdash_{def} ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) := \\ \qquad \forall\ t{:}\ Clock\_cycle. \\ \qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{20..16}) \text{ in} \\ \qquad\qquad A(t+1) = RF(t)\ [rs1(t)] \wedge \\ \qquad\qquad B(t+1) = RF(t)\ [rs2(t)] \wedge \\ \qquad\qquad IR1(t+1) = IR(t) \end{array}$$

Stage instructions which include operations using a class abstraction function, e.g. EX_A, are specified in a similar way except that the class abstraction function *op* is introduced as a part of the predicate parameters.

Continuing with the semantic specification of the control instructions, the ID-stage instructions ID_C of the CONTROL-class (see row ID and column CONTROL in table 1) is specified as follows:

$$\begin{array}{l} \vdash_{def} ID_C\_SPEC\ f_C\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) := \\ \qquad \forall\ t{:}\ Clock\_cycle. \\ \qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}) \text{ in} \\ \qquad\qquad PC(t+1) = f_C\ (PC(t), offset16(t), offset26(t), RF(t)\ [rs1(t)]) \end{array}$$

3.2.4. Phase Level

Similarly to the stage instructions, the predicates for phase instructions are built up of conjunctions of elementary state transitions that 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 (see for example the predicate $\phi_1 ID_A\_SPEC$ below). As a refinement of the ID-stage specifications for the ALU-class, the corresponding ID-phase instructions can be easily specified by the following predicates:

$$\begin{array}{l} \vdash_{def} \phi_1 ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) := \\ \qquad \forall\ \tau{:}Clock\_phase. \\ \qquad\quad IR(\tau+1) = IR(\tau) \end{array}$$

$$\begin{array}{l} \vdash_{def} \phi_2 ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) := \\ \qquad \forall\ \tau{:}Clock\_phase. \\ \qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{20..16}) \text{ in} \\ \qquad\qquad A(\tau+1) = RF(\tau)\ [rs1(\tau)] \wedge \\ \qquad\qquad B(\tau+1) = RF(\tau)\ [rs2(\tau) \wedge \\ \qquad\qquad IR1(\tau+1) = IR(\tau) \end{array}$$

In contrast to the previous example, the stage operation of ID_C is accomplished during the first phase and held in the buffer *BTA*, during the second phase:

$$\vdash_{def} \phi_1 ID_C\_SPEC\ \ f_C\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=$$
$$\forall\ \tau{:}Clock\_phase.$$
$$\text{let } (rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}) \text{ in}$$
$$BTA(\tau+1) = f_C\ (PC(\tau), offset16(\tau), offset26(\tau), RF(\tau)\ [rs1(\tau)])$$

$$\vdash_{def} \phi_2 ID_C\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) :=$$
$$\forall\ \tau{:}Clock\_phase.$$
$$PC(\tau+1) = BTA(\tau)$$

3.2.5. Electronic Block Model

While the abstract levels of our interpreter model are behavioural descriptions, the EBM describes the structure of the hardware at the RT-level (Register-Transfer). However, the visible states and the temporal refinement used are the same as those of the phase level. In addition to the state components, environment signals, such as the clock, interrupt or bus control signals, are made visible and therefore will be involved as part of the specification predicate parameters. The EBM is in general structured hierarchically at the RT-level. At the top most level, the EBM is composed

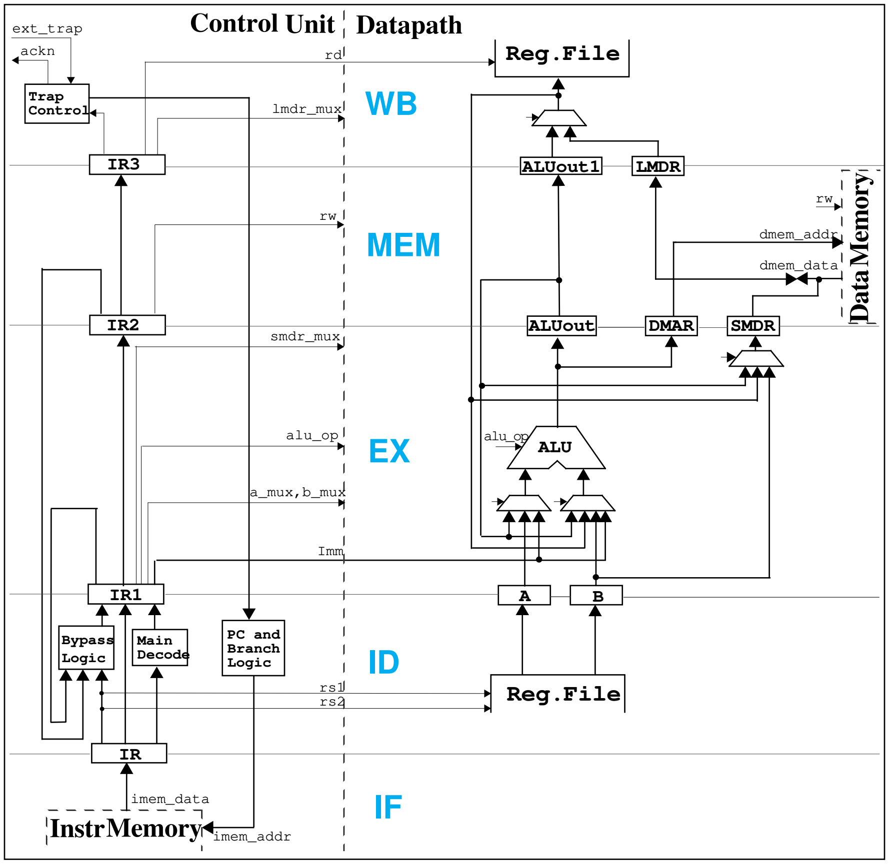

Figure 6. Electronic Block Model of DLX (simplified)

of the RISC processor core and the interfaced instruction and data memories (caches). The processor is conventionally split into a datapath and a control unit. These are themselves compositions of simpler blocks, e.g. register file, arithmetic-logic unit (ALU), multiplexers, pipeline latches, etc., which may again be conjunctions of lower building blocks.

Figure 6 shows a simplified form of the EBM of a DLX processor implementation [28]. The data path and the control unit implement the pipelined execution. They are physically composed of series of functional units and pipeline buffers and are partitioned in the above diagram according to the pipeline stages. In the IF-stage, the instruction memory (cache) is accessed. In the ID-stage, the fetched instructions are decoded, target addresses are computed, the register file is accessed and all control signals, e.g. for bypassing control, are generated. In the EX-stage, the ALU is exercised for arithmetic-logic operations or data address calculation. In the MEM-stage, the data memory (cache) is eventually accessed. Finally, in the WB-stage the computed results or loaded data are put into the register file and all internal and external interrupts occurring during the instruction execution are handled.

Formally, the EBM is specified à la Hanna and Daeche [41] as a complex hierarchy of predicates, which are composed using conjunctions. The input/output lines are universally quantified and the internal lines of the circuit are modelled using existential quantification. The top level implementation of the EBM given in figure 6 looks formally as follows, where the processor predicate is expanded into datapath and control unit:

$\vdash_{def}$ ***EBM*** (*PC, I-MEM, RF, D-MEM, A, B, ALUOUT, ALUOUT1, DMAR, SMDR,*
LMDR, IR, IR1, IR2, IR3, BTA, IAR, ext_trap, ackn, clk1, clk2) :=

$\exists$ *rs1, rs2, rd, Imm, a_mux, b_mux, alu_op, smdr_mux,*
lmdr_mux, imem_addr, imem_data, dmem_addr, dmem_data, rw.

DataPath (*RF, A, B, ALUOUT, ALUOUT1, DMAR, SMDR, LMDR,*
dmem_addr, dmem_data, rs1, rs2, rd, Imm, a_mux,
b_mux, alu_op, smdr_mux, lmdr_mux, clk1, clk2) $\wedge$

Control_Unit (*PC, IR, IR1, IR2, IR3, BTA, IAR, ext_trap, ackn, rw,*
imem_addr, imem_data, rs1, rs2, rd, Imm, a_mux,
b_mux, alu_op, smdr_mux, lmdr_mux, clk1, clk2) $\wedge$

Instr_Memory (*I-MEM, imem_addr, imem_data, clk2*) $\wedge$

Data_Memory (*D-MEM, dmem_addr, dmem_data, rw, clk2*)

Related microprocessor verification works based on the interpreter model describe the EBM using abstract sub-blocks whose implementations are assumed to be correct [5, 49, 77]. In contrast, within our approach the implementation description is completely specified down to the gate level. Since our methodology is embedded within the MEPHISTO [52] verification framework, the formal description of the circuit can either be obtained automatically from an EDIF [31] output of a schematic representation within a CAD tool or from a VHDL description [52]. The sub-blocks of the hierarchical design are broken into elementary library cells whose formal descriptions are contained in a library of rudimentary formal specifications [51].

3.3. Use of the Model Formalization

In general, a formal description of a hardware system could be used for specification, verification, simulation or synthesis [33]. Hence, in addition to the verification intentions, the higher-order logic formalization of our RISC model can be utilized for other purposes such as simulation or formal synthesis, where higher-order logic plays the role of a universal hardware description language. Regarding the use of the model specifications for simulation, Camilleri [18] has shown how higher-order logic specifications can be made executable and run for simulation. This simulation should not replace verification, but rather complement it; by giving the designer more confidence about the specification, against which the implementation will be verified. The formalization of the model could also be used as input for other special tools, e.g. formal synthesis [42]. This approach of formal synthesis incorporates formal verification with the design process. In this sense, our model specifications can also be used during the design process of a RISC processor.

4. Management of the Verification Task

Starting from the architecture of a microprocessor, the aim of a formal processor verification is to show that the instruction set is correctly executed by the hardware. During any clock cycle, the RISC processor can potentially be executing n_s instructions in parallel, in n_s different stages (see figure 7), given that n_s is the pipeline depth. This parallel execution increases the overall throughput of the processor; however no single instruction runs faster, since each instruction is realized by a sequential execution of its stage instructions. In proving the correctness of the RISC processor, we have to therefore prove that each instruction is correctly implemented by the sequential execution of its stage instructions. On the other hand, due to the simultaneous use of shared resources and the existence of data and control dependencies, the stage instructions within the pipeline could interfere with each other, so that semantical inconsistencies could also occur. This fact implies that two orthogonal proofs have to be performed — firstly, the sequential execution of each instruction is correctly implemented by the hardware EBM and secondly the pipelined execution of the instructions is correct. Thus the overall correctness proof is split into two independent steps as follows:

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

$$\vdash \ \mathit{EBM} \Rightarrow \ \mathit{Architecture\ Level} \tag{I}$$

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

$$\mathit{SW\_Constraints,\ EBM} \ \vdash \ \mathit{Correct\_Instr\_Pipelining} \tag{II}$$

The software constraints in (II) 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.

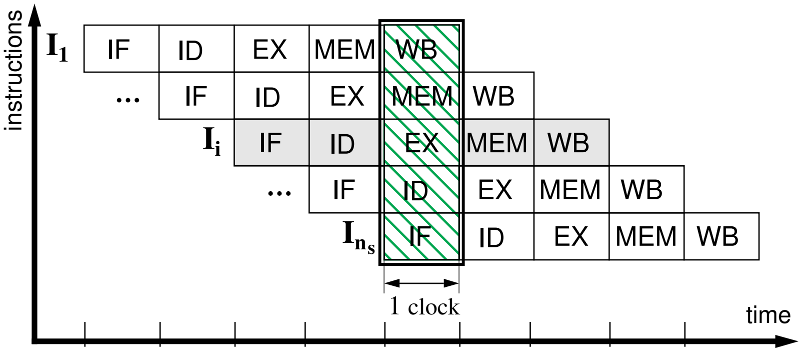

Figure 7. Pipelined Instructions Execution

According to the nature of each of these verification steps, we call step (I) the verification of the semantic correctness and step (II) the verification of the pipeline correctness. These steps are briefly discussed in the following subsections and elaborated later in sections 5 and 6.

4.1. Semantic Correctness

In order to show that the sequential execution of each instruction is correctly implemented by the hardware EBM, we use the higher-order logic specifications and implementations at the various levels of abstraction (cf. section 3.2) and prove the following — *EBM* ⇒ *Phase Level* ⇒ *Stage Level* ⇒ *Class Level*. Later, these proofs are instantiated for each instruction at the architecture level. The verification tasks of the semantic correctness include a proof for every instruction of each interpreter level. However, exploiting the existing similarities between instructions of a given abstraction level this tedious process could be automated using parameterized functions and proof scripts that automatically generate the verification goals and perform the proofs for whole sets of instructions, respectively.

4.2. Pipeline Correctness

The pipeline correctness consists in the proof that all possible combinations of n_s instructions, within the pipeline, are executed correctly. In the RISC literature, the inconsistencies that arise due to the data and control dependencies and the resources contentions that occur in a pipelined execution are called *conflicts*. There are three classes of conflicts (also called *hazards*) namely, *resource*, *data* and *control conflicts* [43]. Since the pipeline correctness is the direct consequence of the absence of all these conflicts, the correctness statement (II) defines the non-existence of these conflicts. The predicate *Correct_Instr_Pipelining* in (II) is hence defined as the following conjunction, where we assign a specific conflict predicate to each kind of conflict, i.e. *Resource_Conflict*, *Data_Conflict* and *Control_Conflict*. Formally:

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

and the pipeline correctness statement (II) can be rewritten as:

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

All these conflict predicates have to be formally specified and should be proven false. Thus the whole correctness proof is tackled by splitting it into three independent parts, each corresponding to one kind of conflict.

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 (as described in section 2.2) is exploited by considering the combinations of few classes instead of combinations of all instructions. Thus, all conflict predicates will be specified at a higher level in terms of class instructions. Furthermore, it will be of a great advantage to closely relate the specifications of these conflicts to the hierarchical levels of our interpreter model, taking the temporal and structural abstractions into account.

4.3. Verification of Specific Hardware Behaviours

A RISC processor generally includes some hardware behaviours whose specifications and implementations are processor specific, such as hardware interrupts, stalls, branch prediction, etc. In contrast to the architecture of the processor core, these specific behaviours cannot be handled mechanically within our methodology since they are highly implementation dependent. Different RISC processors handle interrupts, stalls, freezes and branch prediction in different ways, e.g. for interrupts where the forced jump is to be inserted, the manner in which the interrupted program is restarted, etc. can vary. Therefore, in addition to the specifications of the described model levels (cf. section 3.2), one has to specify the intended (interrupt, stall, freeze or branch prediction) behaviour formally in form of a predicate, whose correctness has to be implied from the implementation EBM. Furthermore, the specification should take into account the pipelined behaviour of instructions executions.

In general, such hardware behaviours could be grouped into two groups, namely

1. hardware used for conflict resolution (resource, data or control), e.g. branch prediction, bypassing logic, etc., which depend on the internal state of the processor, and
2. hardware used for specific features, e.g. interrupts, stalls, etc., which in addition depend on the external environment of the processor.

For the former group, the specification predicates of the implemented behaviour are used for the proof of the pipeline correctness and their verification is implicitly included in step (II) of the correctness statement (cf. section 6.3.2). The verification of the latter group is handled separately in addition to the steps (I) and (II). For example, let *INTR_SPEC* be a predicate that describes the behaviour of the implemented hardware interrupt of a specific processor and which ensures that no resource, data or control conflicts occur when the linear pipeline flow is interrupted (e.g. proper saving and recovery of the processor state before and after the interrupt handling), then the goal to be proven for the specific hardware interrupt behaviour is:

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

5. Semantic Verification

In verifying the semantic correctness of RISC instructions, we consider the fact that the execution of each RISC instruction is realized by the sequential execution of instructions at lower abstraction levels having different time granularities. Hence, we first hierarchically prove the correctness of the class level, by using the notion of instruction classes and the hierarchical verification model (cf. section 2.2) i.e.:

$$EBM \Rightarrow Phase\ Level \Rightarrow Stage\ Level \Rightarrow Class\ Level$$

and then through instantiation, we show the correctness of the architecture level. Corresponding to the abstraction levels, this proof is broken into the following steps:

$Stage\ Level \Rightarrow Class\ Level$,
$Phase\ Level \Rightarrow Stage\ Level$ and
$EBM \Rightarrow Phase\ Level$

Due to this structuring of the verification task, the verification goals at different levels are simple and show some similarities. The proofs are managed easier and general proof strategies could be developed. In many aspects, this verification process is similar to that used by Windley [77] for the verification of microprogrammed processors.

The description of the goals and their associated proofs are accomplished automatically using generic functions and tactics, respectively. Further, each abstraction level can be verified independently, so that a designer is able to successively refine and verify the design. In the following, we present the verification process at each level of abstraction and we then show how the instantiations are handled. The verification techniques described will be illustrated by some simple examples based on the DLX processor.

5.1. Class Level Verification

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, i.e.:

$$IF\_SPEC \wedge ID\_SPEC \wedge ... \wedge WB\_SPEC \Rightarrow CLASS\_SPEC$$

Since the specifications of the class and stage levels use different time granularities, the verification goal should include the temporal abstraction function and context parameters (as discussed in section 3.1). Hence, within the verification goal, the abstract specification (here the class instruction) should be extended in such a way that for each state component of the class specification formula a context parameter is introduced and the time abstraction function is applied to the abstract time variables (here instruction cycles u). The implementation dependent context parameters are introduced as existentially quantified variables that have to be instantiated appropriately later during the proof. The temporal abstraction function *Time_abs* (as presented in section 3.1) will be instantiated with the corresponding pipeline depth n_s and is applied to the different context variables.

Using the formal specifications of the class and the stage instructions, as described in sections 3.2.2 and 3.2.3, respectively, the verification goal for the ALU-class example looks formally as follows:

$$
\begin{array}{l}
\vdash \forall\, op. \\
\quad IF_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \ \wedge \\
\quad ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \ \wedge \\
\quad EX_A\_SPEC\ \ f_A\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \ \wedge \\
\quad MEM_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \ \wedge \\
\quad WB_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\
\quad \Rightarrow \exists\, C_1\ C_2\ C_3 \colon Context. \\
\qquad \text{let } f_t = (Time\_abs\ n_s) \text{ in} \\
\qquad\quad \forall u\colon Instr\_cycle. \\
\qquad\qquad (RF \circ f_t\ C_1)\ (u+1)\ [(rd \circ f_t\ C_2)\ (u)] = \quad ((RF \circ f_t\ C_3)\ (u)\ [(rs1 \circ f_t\ C_2)\ (u)])\ op \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ((RF \circ f_t\ C_3)\ (u)\ [(rs2 \circ f_t\ C_2)\ (u)])
\end{array}
$$

In order to avoid the burden of setting such complex verification goals (which may be error prone), we have developed a parameterized function which automatically generates the required goals given the pipeline depth and the corresponding specification predicates as parameters. This function takes into account the time abstraction function and extracts in an intelligent way the needed context variables C_i from the abstract specification. Let $\mathcal{G}$ be this goal setting function. Using the following parametrization for $\mathcal{G}$:

$$\mathcal{G}\ (n_s,\ CONTROL\_SPEC,\ [IF_C\_SPEC,\ ID_C\_SPEC])$$

the verification goal of the CONTROL-class which has to be implied from the conjunction of an IF and an ID-stage instructions is generated automatically as:

$$
\begin{array}{l}
\vdash \forall\, f_C. \\
\quad IF_C\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \wedge \\
\quad ID_C\_SPEC\ \ f_C\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\
\qquad \Rightarrow\ \exists\, \mathrm{C}_1\ C_2\ \mathrm{C}_3\ C_4 \colon Context. \\
\qquad\quad \text{let } f_t = (Time\_abs\ n_s) \text{ in} \\
\qquad\qquad \forall u\colon Instr\_cycle. \\
\qquad\qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}\,) \text{ in} \\
\qquad\qquad\qquad (PC \circ f_t\ C_1)\ (u+1) = \\
\qquad\qquad\qquad\qquad\qquad f_C\ ((PC \circ f_t\ C_2)\ (u),\ (offset16 \circ f_t\ C_4)\ (u), \\
\qquad\qquad\qquad\qquad\qquad\quad (offset26 \circ f_t\ C_4)\ (u),\ (RF \circ f_t\ C_3)\ (u)\ [(rs1 \circ f_t\ C_4)\ (u)])
\end{array}
$$

The universal quantification of the class abstraction functions op and f_C over the entire verification goal expresses the generality of the theorem that is to be proven. Therefore, the obtained theorems and the corresponding abstraction functions can be instantiated for special architectural instructions.

For the proof of the class level, we use a general common tactic with the following parameters: the pipeline depth n_s, the class and corresponding stage specification predicates and a list of the needed context parameters including read/write information and an indication of the corresponding pipeline stage. This tactic is mainly based on breaking down the structural abstraction by splitting the conjunctions of the stage instructions, explicitly instantiating the existentially quantified context variables, expanding the specification predicates of the stage instructions, resolving the let terms, mapping the time variables of the class level to those of the stage level using the temporal abstraction function, and finally applying arithmetical and logical simplifications and several rewritings. Let $\mathcal{T}$ be this tactic, for the above goal of the ALU-class verification, we use the following parameters for $\mathcal{T}$:

$$
\begin{array}{l}
\mathcal{T}(n_s, \\
\quad ALU\_SPEC, \\
\quad [IF_A\_SPEC, ID_A\_SPEC, EX_A\_SPEC, MEM_A\_SPEC, WB_A\_SPEC], \\
\quad [[write,\ WB], [read,\ IF], [read,\ ID]]\ )
\end{array}
$$

which automatically yields the correctness proof of the ALU-class instruction. The context tuples given within the parameters of $\mathcal{T}$ 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 in the above verification goal for the ALU-class is applied to it using the context parameter $C_3 = [read,\ ID]$.

5.2. Stage Level Verification

The correctness proof between the phase and the stage level is done in a manner similar to the previous section, by proving that each stage instruction implementation is implied by the conjunction of the corresponding phase instructions:

$$\phi_1 ID\_SPEC \wedge \ldots \wedge \phi_{n_p} ID\_SPEC \Rightarrow ID\_SPEC$$

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, goal setting and proof mechanisms are used. However, the appropriate parameters, e.g. number of clock phases, phase identifiers, etc., have to be set accordingly.

The verification goals of the stage level should involve the extension of the abstract specification (here the stage instruction) by the temporal abstraction function and the appropriate context variables which are included for each state component. Furthermore, the time abstraction function *Time_abs* should be instantiated with the corresponding number of clock phases n_p. Using the specifications of the stage and phase levels, as described in sections 3.2.3 and 3.2.4, respectively, the verification goal for the ID-stage of the ALU-class is given below:

$$
\boxed{
\begin{array}{l}
\vdash\ \ \phi_1 ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, BTA) \wedge \\
\qquad \phi_2 ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, BTA) \\
\qquad\quad \Rightarrow \exists\ C_1\ C_2\ C_3\ C_4\ C_5\text{: } Context. \\
\qquad\qquad\quad \text{let}\ f_t = (Time\_abs\ n_p)\ \text{in} \\
\qquad\qquad\qquad \forall t\text{: } Clock\_cycle. \\
\qquad\qquad\qquad\quad \text{let}\ (rs1 = [IR]_{25..21} \wedge rs2 = [IR]_{22..16})\ \text{in} \\
\qquad\qquad\qquad\qquad (A \circ f_t\ C_1)\ (t{+}1) = (RF \circ f_t\ C_3)\ (t)\ [(rs1 \circ f_t\ C_4)\ (t)] \wedge \\
\qquad\qquad\qquad\qquad (B \circ f_t\ C_2)\ (t{+}1) = (RF \circ f_t\ C_3)\ (t)\ [(rs2 \circ f_t\ C_4)\ (t)] \wedge \\
\qquad\qquad\qquad\qquad (IR1 \circ f_t\ C_5)\ (t{+}1) = (IR \circ f_t\ C_4)\ (t)
\end{array}
}
$$

Such verification goals for the stage level can also be set automatically using the presented function $\mathcal{G}$ with the appropriate parameters. For example, through the following parametrization of $\mathcal{G}$:

$$\mathcal{G}\,(n_p, f_C, ID_C\_SPEC, [\phi_1 ID_C\_SPEC, \phi_2 ID_C\_SPEC])$$

the following verification goal for the ID-stage of the CONTROL class is generated:

$$\boxed{\begin{array}{l}
\vdash \forall f_C. \\
\quad \phi_1 ID_C\_SPEC\ \ f_C\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \wedge \\
\quad \phi_2 ID_C\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \\
\qquad\quad \Rightarrow \exists\, C_1\, C_2\, C_3\, C_4\!: Context. \\
\qquad\qquad\quad \text{let } f_t = (Time\_abs\ n_p) \text{ in} \\
\qquad\qquad\qquad \forall t\!: Clock\_cycle. \\
\qquad\qquad\qquad\quad \text{let } (rs1 = [IR]_{25..21} \wedge offset16 = [IR]_{15..0} \wedge offset26 = [IR]_{25..0}\,) \text{ in} \\
\qquad\qquad\qquad\qquad (PC \circ f_t\, C_1)\,(t{+}1) = \\
\qquad\qquad\qquad\qquad\qquad\qquad f_C\,((PC \circ f_t\, C_2)\,(t), (offset16 \circ f_t\, C_4)\,(t), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (offset26 \circ f_t\, C_4)\,(u), (RF \circ f_t\, C_3)\,(t)\,[(rs1 \circ f_t\, C_4)\,(t)])
\end{array}}$$

For the proof of the stage level, the same parameterized tactic $\mathcal{T}$ used for the class level verification is now applied with the following parameters: the number of clock phases n_p, the predicate of the stage instruction, the corresponding phase instruction predicates and an explicit list of the needed context tuples. For the above ID-stage verification goal example of the ALU class, the proof is automatically achieved by applying the tactic $\mathcal{T}$ with the following parametrization:

$$\begin{array}{l}
\mathcal{T}(\ n_p, \\
\quad ID_A\_SPEC, \\
\quad [\phi_1 ID_A\_SPEC, \phi_2 ID_A\_SPEC], \\
\quad [[write, \phi_2], [write, \phi_2], [read, \phi_2], [read, \phi_1], [write, \phi_2]])
\end{array}$$

The context variables required for the proof are easily derived from the implemented pipeline structure (table 1), e.g. since the B-register in the ID-stage of the ALU-class is written in phase 2, the time abstraction function in the verification goal of the ID_A stage instruction is applied to it, using the context variable $C_2 = [write, \phi_2]$.

5.3. Phase Level Verification

The phase level lies directly above the EBM. This step of the verification is different from the previous ones since the EBM is a structural specification, while the phase level is a behavioural one. However, due to the advantage of having used the hierarchical interpreter model, we only have to show the correctness of a reduced number of phase instructions, built up of simple transitions as seen in section 3.2.4. Although the specification of EBM is quite complex, a large amount of automation has been achieved in the domain of hardware verification at the RT-level, e.g. MEPHISTO [52]. The goal to be proved is successively broken down into a number of smaller subgoals which can then be solved more or less automatically by the MEPHISTO verification framework.

For the correctness proof at the phase level, it is to be noted that phase instructions including class abstraction functions cannot be proven correct for every possible instance of the abstraction function, since the implementation EBM only provides the concretely implemented ones, e.g. the operator *op* cannot be instantiated for floating point operations if no floating point arithmetic is provided by the actual hardware. Hence, according to the implementation EBM, instead of a general theorem including an universal quantification over the class abstraction function, we rather prove instantiated phase instructions. In order to ease the verification of the relatively large number of very similar phase instructions, we have developed an appropriate function which automatically generates the verification goals for the correctness of the phase level. The parameters for this function are: the predicate of the phase instruction, the corresponding clock phase identifier, the predicate of the implementation EBM and a list $\mathcal{L}$ representing the instances of the class abstraction function that are intended by the architecture, e.g. for arithmetic-logic operations $\mathcal{L} = [add, sub, or, shl, ...]$. According to the number of elements (if any) in the list $\mathcal{L}$, this function generates the appropriate number of verification goals for instantiated phase instructions. In the case of phase instructions that do not include class abstraction, this list is empty and only one goal is generated. Further, using the clock phase identifier, this goal generation function ensures that the input lines for the clock phases within the EBM predicate are set correctly with respect to the phase instruction that is to be implied, e.g. for a two phased clock, the clock signals are set for phase 1 as: $clk1 = \mathsf{T}$ and $clk2 = \mathsf{F}$. Let $\mathscr{g}$ be this goal generation function, using the specifications of the phase level and EBM, as described in sections 3.2.4 and 3.2.5, respectively, the following parametrization example of $\mathscr{g}$:

$$\mathscr{g}(\phi_2 ID_A\_SPEC, \phi_2, EBM, [])$$

generates the following verification goal for the phase instruction $\phi_2 ID_A$ of the ID-stage of the ALU-class:

$$\boxed{\begin{array}{l} \vdash\ EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{\mathsf{F}}, \mathbf{\mathsf{T}}) \\ \qquad \Rightarrow \phi_2 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{array}}$$

The phase instruction $\phi_2 ID_A$ does not include any class abstraction function and therefore the list $\mathcal{L}$ is empty. Another example for the phase level verification is the phase 1 of the ID-stage of CONTROL instructions $\phi_1 ID_C$. In this case the class abstraction function f_C should be instantiated for the provided control instructions, e.g. jump immediate (JMP), jump register indirect (JR) and branch-on-zero (BRZ). Therefore, the function $\mathscr{g}$ is parameterized as follows:

$$\mathscr{g}(\phi_1 ID_C\_SPEC, \phi_1, EBM, [f_{JMP}, f_{JR}, f_{BRZ}])$$

to generate the following three goals:

$$\boxed{\begin{array}{l} \vdash\ EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{\mathsf{T}}, \mathbf{\mathsf{F}}) \\ \qquad \Rightarrow \phi_1 ID_C\_SPEC\ \ f_{JMP}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{array}}$$

$$\boxed{\begin{array}{l} \vdash\ EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{\mathsf{T}}, \mathbf{\mathsf{F}}) \\ \qquad \Rightarrow \phi_1 ID_C\_SPEC\ \ f_{JR}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{array}}$$

$$\boxed{\begin{array}{l} \vdash\ EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{\mathsf{T}}, \mathbf{\mathsf{F}}) \\ \qquad \Rightarrow \phi_1 ID_C\_SPEC\ \ f_{BRZ}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{array}}$$

Since neither structural, data nor temporal abstractions exist between the EBM and the phase level, and the specifications within this goal are at the RT-level, the proof of the phase level could be done automatically using an appropriate general proof schema based on MEPHISTO. Hence, we have developed one general parameterized tactic which proves the correctness of all phase instructions automatically. This tactic is based on the several tactics available in MEPHISTO which automatically expand the specification predicates, flatten the hierarchical description of the EBM, eliminate combinatorial line variables, etc.

5.4. *Instantiations*

In this last part of the verification task, we deal with the correctness proofs at the architectural level. This is obtained by simply instantiating the proven theorems at the class and stage levels and using the already instantiated theorems at the phase level. In the following, we will trace the instantiation procedure by means of the ADD-instruction.

Starting from the architectural level, we first show the equivalence between each particular instruction specification and the related abstract class specification which has been instantiated appropriately (step (4) in figure 3). For the ADD-Instruction we obtain the following theorem:

$$\begin{array}{l} \vdash ADD\_SPEC\ (PC, RF, I\text{-}MEM, D\text{-}MEM) = \\ \quad ALU\_SPEC\ \ \mathbf{add}\ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array}$$

Furthermore, since the correctness proofs at the class level involve an universal quantification of the class abstraction functions, we are able to set an explicit function for the class abstraction and obtain a theorem for a particular instruction of the architecture level. For example, we instantiate the proven theorem for the ALU-class (cf. section 5.1) with the operator constant *add* and obtain the following theorem:

$$\begin{array}{l} \vdash IF_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad EX_A\_SPEC\ \ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad MEM_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad WB_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\ \quad \Rightarrow ALU\_SPEC\ \ \mathbf{add}\ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array}$$

From this and the previous theorems, the correctness of the ADD-instruction from the stage level is easily shown through simple rewriting, i.e:

$$\begin{array}{l} \vdash IF_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad EX_A\_SPEC\ \ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad MEM_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)\ \wedge \\ \quad WB_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \\ \quad \Rightarrow ADD\_SPEC\ \ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{array}$$

At the stage level, only those instructions which include a class specific parameter need to be instantiated, since all other instructions are already proven correct from the phase level. The related theorems are gained in a manner similar to that of the class level, by simple instantiation. For the above example of the ADD-instruction, the EX-stage EX_A is simply proven correct by instantiating the operator *op* using the special *add* operator within the general theorem obtained, i.e:

$$\begin{aligned} \vdash\ & \phi_1 ID_A\_SPEC\ (A, B, \ldots, PC, RF, \ldots, BTA) \wedge \\ & \phi_2 ID_A\_SPEC\ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, BTA) \\ & \quad \Rightarrow EX_A\_SPEC\ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots) \end{aligned}$$

Since the correctness of all phase instructions, including those of the EX-stage which involves the implemented *add* operator, has been shown from the implementation EBM (cf. section 5.3), we use the proven theorems for the phase instructions, e.g.:

$$\begin{aligned} \vdash\ & EBM\ (PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathsf{F}, \mathsf{T}) \\ & \quad \Rightarrow \phi_2 ID_A\_SPEC\ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA) \end{aligned}$$

to obtain the correctness of the required stage instructions IF_A, ID_A, EX_A, MEM_A and WB_A from the implementation EBM. For example, the instantiated EX_A stage instruction is:

$$\begin{aligned} \vdash\ & EBM\ (PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, clk1, clk2) \\ & \Rightarrow EX_A\_SPEC\ \mathbf{add}\ (A, B, \ldots, PC, RF, \ldots, IR, \ldots)) \end{aligned}$$

The correctness of the ADD-instruction from the hardware EBM can be derived through transitivity:

$$\begin{aligned} \vdash\ & EBM\ (PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, clk1, clk2) \\ & \Rightarrow ADD\_SPEC\ (PC, I\text{-}MEM, RF, D\text{-}MEM) \end{aligned}$$

5.5. *Summary*

Having proven the correctness of the class, stage and phase levels and after making the appropriate instantiations of the obtained theorems for specific architectural instructions, we have deduced the correctness proof of the architectural level from the hardware implementation EBM:

$$\vdash\ EBM \Rightarrow Architecture\ Level$$

6. Pipeline Verification

In this section, we focus our attention on the pipeline verification. As discussed in section 4.2, this corresponds to the verification of the resource, data and control pipeline conflicts. Each of these conflicts has to be specified formally as a predicate whose negation is to be proved. Since each conflict can be handled independently, we formalize and describe the proof techniques for a specific conflict in the subsections dedicated to each of them. The proof techniques that are given are automated and moreover 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.

In order to simplify the formalization and proof of pipeline conflicts, they will be specified hierarchically according to the abstraction levels of our RISC model. Additionally, the existence of multiple instructions in the pipeline can be formalized by predicates which we call the *multiple* *conflict predicates*. These multiple conflicts are further defined at lower levels in terms of conflict predicates between pairs of instructions which are called the *dual conflict predicates*. These notions will be clarified in the subsections to follow.

6.1. Formal Definitions

In this section, we briefly introduce some new types, functions and predicates, that are useful for formalizing pipeline conflicts. According to our hierarchical model, we define for each abstraction level a set of enumeration types for the processor specific instructions, resources and pipeline characteristics, i.e. pipeline stages or clock phases. Referring to the pipeline structure in table 1, the required enumeration types are defined for the DLX example with the following arguments:

- types for pipeline stages and clock phases:
  - $pipeline\_stage = IF \mid ID \mid EX \mid MEM \mid WB$
  - $clock\_phase = \phi_1 \mid \phi_2$
- types for the set of all instructions at each abstraction level:
  - $class\_instruction = ALU \mid LOAD \mid STORE \mid CONTROL$
  - $stage\_instruction = IF_X \mid ID_X \mid ID_C \mid EX_A \mid \ldots \mid MEM_S \mid \ldots \mid WB_L$
  - $phase\_instruction = \phi_1 IF_X \mid \ldots \mid \phi_1 EX_A \mid \ldots \mid \phi_1 WB_L \mid \phi_2\, IF_X \mid \ldots \mid \phi_2 WB_L$
- types for resources (related to the structural abstraction, where CL, SL and PL stand for Class Level, Stage Level and Phase Level, respectively):
  - $CL\_resource = PC \mid RF \mid I\text{-}MEM \mid D\text{-}MEM$
  - $SL\_resource = PC \mid RF \mid I\text{-}MEM \mid \ldots \mid IR \mid A \mid B \mid ALUOUT \mid DMAR \mid \ldots$
  - $PL\_resource = PC \mid RF \mid I\text{-}MEM \mid \ldots \mid IR \mid A \mid B \mid ALUOUT \mid \ldots \mid BTA$

Since the arguments of these enumeration types are processor specific, they have to be defined for each RISC differently. Except these type definitions, all needed information about the specific processor that is to be verified are explicitly extracted from the formal specifications of the model levels (cf. section 3.2).

For the specification of the conflict predicates, we also define the following functions and predicates:

- abstraction functions, which either compute higher level instructions from lower ones or extract lower level instructions from higher ones[6]:
  - *ClassToStage*: ((*pipeline_stage*, *class_instruction*)$\rightarrow$ *stage_instruction*)
  - *StageToClass*: (*stage_instruction* $\rightarrow$ *class_instruction*)
  - *StageToPhase*: ((*clock_phase, stage_instruction*) $\rightarrow$ *phase_instruction*)
  - *PhaseToStage*: (*phase_instruction* $\rightarrow$ *stage_instruction*)

  e.g. *ClassToStage* (*ID*, *CONTROL*) = ID_C, *PhaseToStage* ($\phi_2 EX_A$) = EX_A.

---

[6] The notation "f: (α, β, ...) $\rightarrow$ δ" means that the function f has arguments of types α, β, ... and a range of type δ.

- 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:

  Stage_Rank: (*pipeline_stage* → *num*)
  Phase_Rank: (*clock_phase* → *num*)

  e.g. *Stage_Rank* (*ID*) = 1, *Phase_Rank* (ϕ_1) = 0. These functions are needed to express the sequential order of the execution of stage and phase instructions.

- 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* which means that the resource *PC* is used (written) by the stage instruction ID_C.

- predicates that imply that a given resource is read (*domain*) or written (*range*) [50] 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*)

  e.g. *Stage_Domain* (*ALU*, *ID*, *RF*) = *True*, *Phase_Range* (ID_C, ϕ_2, *D-MEM*) = *False* which means that the register file *RF* is read by the ALU-class instruction at the ID-stage and that the data memory *D-MEM* is not written by the stage instruction ID_C at the second clock phase, respectively (refer also to table 1).

The Predicates *Stage/Phase_Used*, *Stage/Phase_Domain* and *Stage/Phase_Range* are automatically extracted from the specifications of the class, stage and phase level instructions at the clock cycle and clock phase time granularities, respectively (refer to section 3.2). Each of these predicates is generated as a theorem for the given combination of class, stage or phase instruction, pipeline stage or clock phase and resources corresponding to a particular level. 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. The process of extracting the above predicates is done completely automatically using four functions — one for *Stage_Used*, one for *Phase_Used*, one for *Stage_Range/Domain* and one for *Phase_Range/Domain*. These functions use the defined processor specific types and the formal specifications of the class, stage and phase levels (as given in section 3.2, e.g. *ALU_SPEC*, ID_C*_SPEC*, etc.) and generate the required theorems from the formal specifications.

6.2. Resource Conflicts

Resource conflicts (also called *structural hazards* [43, 50, 57, 67] or *collisions* [50, 67]) 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 subsections to follow, we will first formally specify the resource conflicts and then discuss the correctness proof issues.

6.2.1. Resource Conflict Specification

Referring to the hierarchical RISC model, 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. In the following subsections, the specification of the resource conflicts is presented hierarchically at the class, stage and phase levels. Other specification forms for resource conflicts diverging from the one to follow are of course possible, e.g. [72] where a compact formalization for post-design verification is presented.

Class Level Conflicts. The resource conflict predicate *Resource_Conflict*, as mentioned in section 4.2, is equivalent to a multiple conflict between the maximal number of class instructions that occur in the pipeline, i.e.:

$$\boxed{\vdash_{def} \mathit{Resource\_Conflict} := \mathit{Multiple\_Res\_Conflict}\ (I_1, \ldots, I_{n_s})}$$

This *Multiple_Res_Conflict* predicate is true if any pair of the corresponding stage instructions compete for one resource (see hatched box in figure 8). 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 the conflicts between a pair of stage instructions (dual stage conflicts). Using the function *ClassToStage*, the multiple resource conflict is specified formally in terms of dual conflicts as follows (where the index i for ψ_i represents the related pipeline stage, i.e. $\psi_1 = IF$, $\psi_2 = ID$, $\psi_3 = EX$, etc.):

$$\vdash_{def} \mathit{Multiple\_Res\_Conflict}\ (I_1, \ldots, I_{n_s}) := \bigvee_{\substack{i,\, j \\ (i,\, j = 1 \ldots n_s) \\ (i < j)}} \mathit{Dual\_Stage\_Conflict}\ (\mathit{ClassToStage}\ (\psi_i, I_{n_s-i+1}), \mathit{ClassToStage}\ (\psi_j, I_{n_s-j+1}))$$

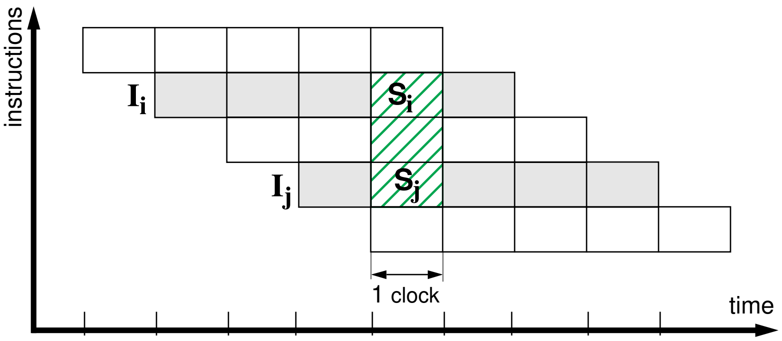

Figure 8. Stage Resource Conflict

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 7), we should ensure that the corresponding stages are of different logical types. Using the function *Stage_Type* and the predicate *Stage_Used*, the *Dual_Stage_Conflict* predicate is specified formally as follows:

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

Looking closer, since a multi-phased non-overlapping clock is used, 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 (figure 9).

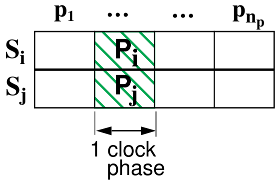

Figure 9. Phase Resource Conflict

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, i.e.:

$$\vdash_{def} \mathit{Dual\_Stage\_Conflict} := \mathit{Multiple\_Phase\_Conflict}\ (S_i, S_j)$$

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. Using the function *StageToPhase*, the multiple phase conflict is specified formally as follows (where the index *k* in φ_k represents a specific clock phase, i.e. $\varphi_1 = \phi_1$, $\varphi_2 = \phi_2$):

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

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 9) 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 defined as follows:

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

6.2.2. Resource Conflict Verification

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:

$$\vdash \quad \forall\, I_1 \ldots I_{n_s} : class\_instruction\,.\; \neg\; Multiple\_Res\_Conflict\;(I_1, \ldots, I_{n_s})$$

Using the definition of *Multiple_Res_Conflict*, the expansion of this goal at the stage level yields a case explosion since for each permutation of n_s class instructions, one has to perform the conflict checks over all possible combinations of dual conflicts (represented by the big disjunction in the specification of *Multiple_Res_Conflict*). Taking advantage of the fact that most of the stage instructions are shared by many class instructions, this complex goal can be simplified by managing the proof in two steps as follows:

1. we prove that dual conflicts cannot occur:

$$\vdash \quad \forall\, S_i\, S_j : stage\_instruction\,.\; \neg\; Dual\_Stage\_Conflict\;(S_i, S_j)$$

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

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

Since the dual conflict predicate, which ranges over all stage instruction pairs, 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 the CONTROL-class is output as follows:

$$(S_i = IF_X), (S_j = ID_C), (r = PC) \vdash \mathsf{F}$$

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:

$$\vdash \neg\; Multiple\_Phase\_Conflict\;(IF_X, ID_C, PC)$$

Using the definition of *Multiple_Phase_Conflict*, this goal is expanded in terms of dual phase conflicts and one obtains either *True* (which means conflict freedom) or a number of subgoals of the form:

$$(P_i = \phi_k IF_X), (P_j = \phi_k ID_C), (r = PC) \vdash \mathsf{F}$$

In this case, the resource conflict remains and the implementation EBM has to be changed appropriately, e.g. by using an additional buffer or splitting the clock cycle into more phases. Furthermore, since the phase level involves all resources of the machine, this result could also be reached by a systematic check of all resource conflicts at the phase level. This is then done by setting the following goal:

$$\vdash \quad \forall\, P_i\, P_j : phase\_instruction\, . \quad \neg\ Dual\_Phase\_Conflict\ (P_i, P_j)$$

However, due to the large number of resources and phase instruction combinations, the proof is very time and memory consuming, but tractable.

To summarize, given an adequate implementation EBM which ensures that no resource is mutually used by either the class, stage and phase instructions in simultaneous execution, respectively, we prove for all instruction combinations and resources of the actual machine that no resource conflicts occur, i.e.

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

6.3. Data Conflicts

Data conflicts (also called *data hazards* [43, 50, 67], *timing hazards* [57], *data dependencies* [35] or *data races* [32]) arise when an instruction depends on the results of a previous instruction. The term data refers either to the contents of some register within the processor or to the contents 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) [35, 43, 50, 67] (also called destination source (DS), source destination (SD) and destination destination (DD) conflicts [57]). 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

The RAW conflict is the most frequent data conflict kind. The WAR and WAW conflicts, however, are less severe and rarely occur except in some special cases. Since the semantics of these data conflicts have similar forms, it is expected that their formal specifications and proofs are also similar, hence a general formalization and verification method could be given. For illustration purposes, in the rest of this section we will mainly focus on RAW data conflicts and then transfer the obtained results to WAR and WAW data conflicts.

6.3.1. 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 hierarchically at the class, stage and phase levels, as described in the next subsections. Other variant specification forms for data conflicts, which are more useful for post-design verification purposes are given in [72].

Class Level Conflicts. Considering a full pipeline (see figure 7), 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 thus 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 $t_1^0 \ldots t_{n_s}^0$ [7], *i.e.*:

$$\vdash_{def} Data\_Conflict := Multiple\_Data\_Conflict\ (I_1, \ldots, I_{n_s})$$

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:

$$\begin{aligned}&\vdash_{def} Multiple\_Data\_Conflict\ (I_1, \ldots, I_{n_s}) := \\ &\quad \exists\, t_1^0 \ldots t_{n_s}^0 : Clock\_cycle. \\ &\qquad \bigvee_{\substack{i,j \\ (i,j = 1 \ldots n_s) \\ (i<j)}} Dual\_Data\_Conflict\ ((I_i, t_i^0), (I_j, t_i^0 + j - 1))\end{aligned}$$

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_i^0 and t_j^0, 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:

$$\begin{aligned}&\vdash_{def} Dual\_Data\_Conflict\ ((I_i, t_i^0), (I_j, t_j^0)) := \\ &\quad \exists\, r : CL\_resource\,. \\ &\qquad Stage\_Data\_Conflict\ ((I_i, t_i^0), (I_j, t_j^0), r)\end{aligned}$$

Stage Level Conflicts. Let I_i be an instruction that is issued into the pipeline at time t_i^0 and *writes* a given resource r at t_i^u ($t_i^0 \leq t_i^u$). Let I_j be another instruction that is issued at later time t_j^0, i.e. ($t_i^0 < t_j^0$) and *reads* the same resource r at t_j^u. A RAW data conflict occurs when the resource r is read by I_j *before* (and not *after*) this resource is written by the sequentially previous instruction I_i (figure 10). 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

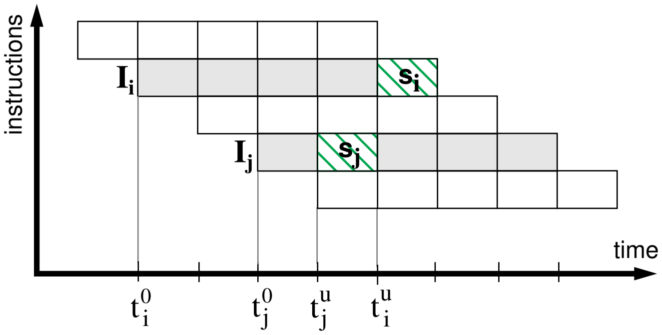

Figure 10. RAW Data Conflict

---

[7] We assume a linear pipelining of instructions, i.e. no pipeline freeze or stall exist, as far as data conflicts are concerned. The use of pipeline stalls or freezes is handled as a specific hardware behaviour apart as described in section 4.3.

happen, the use time points t_i^u and t_j^u are equal to $(t_i^0 + \theta(s_i))$ and $(t_j^0 + \theta(s_j))$, respectively (where the symbol θ represents the function *Stage_Rank*, which computes the ordinal value of a given pipeline stage (cf. section 6.1)). Hence, the timing condition for the RAW conflict, i.e. $(t_j^u \leq t_i^u)$, is equivalent to $(t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))$.

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* $((I_i,\ t_i^0), (I_j,\ t_j^0), r)$:=
$\exists\ s_i\ s_j$: *pipeline_stage* .
$(0 < (t_j^0 - t_i^0)) \wedge$
$((t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))) \wedge$
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* $((I_i,\ t_i^0), (I_j,\ t_j^0), r)$:=
$\exists\ s_i\ s_j$: *pipeline_stage* .
$(0 < (t_j^0 - t_i^0)) \wedge$
$((t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))) \wedge$
Stage_Domain $(I_i, s_i, r) \wedge$
Stage_Range (I_j, s_j, r)

$\vdash_{def}$ *Stage_WAW_Conflict* $((I_i,\ t_i^0), (I_j,\ t_j^0), r)$:=
$\exists\ s_i\ s_j$: *pipeline_stage* .
$(0 < (t_j^0 - t_i^0)) \wedge$
$((t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))) \wedge$
Stage_Range $(I_i, s_i, r) \wedge$
Stage_Range (I_j, s_j, r)

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. $t_i^u = t_j^u$. In this situation, the data conflict should be examined at the phase level.

Phase Level Conflicts. 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 11, 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. $(\tau_i^u \geq \tau_j^u)$. Since instructions at the phase level are executed purely in parallel, they all have the same issue time $\tau_i^0 = \tau_j^0 = \tau^0$ (figure 11), the timing condition $(\tau_i^u \ \geq\ \tau_j^u)$ is equivalent to $(\tau^0 + \xi(p_i)) \geq (\tau^0 + \xi(p_j)) = (\xi(pi) \geq \xi(p_j))$, where the symbol ξ represents the function *Phase_Rank* which computes the ordinal value of the clock phase (cf. section 6.1). Using the functions *Stage_Rank*, *Phase_Rank* and *Stage_Type* (represented by the symbols θ, ξ and ϑ, respectively) and the predicates *Phase_Domain* and *Phase_Range*, the phase level RAW data conflict predicate is formally given as follows:

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

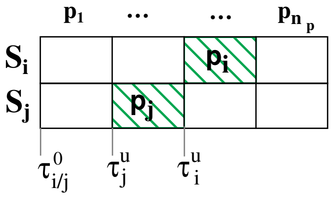

Figure 11. Phase RAW Data Conflict

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

$$\begin{aligned}
&\vdash_{def} \mathit{Phase\_WAR\_Conflict}\ (S_i, S_j, r) := \\
&\qquad \exists\ p_i\, p_j : \mathit{clock\_phase}. \\
&\qquad\quad (\xi(p_j) < \xi(p_i)) \wedge \\
&\qquad\quad (\theta(\vartheta(S_j)) < \theta(\vartheta(S_i))) \wedge \\
&\qquad\quad \mathit{Phase\_Range}\ (S_i, p_i, r) \wedge \\
&\qquad\quad \mathit{Phase\_Range}\ (S_j, p_j, r)
\end{aligned}$$

$$\begin{aligned}
&\vdash_{def} \mathit{Phase\_WAW\_Conflict}\ (S_i, S_j, r) := \\
&\qquad \exists\ p_i\, p_j : \mathit{clock\_phase}. \\
&\qquad\quad (\xi(p_j) < \xi(p_i)) \wedge \\
&\qquad\quad (\theta(\vartheta(S_j)) < \theta(\vartheta(S_i))) \wedge \\
&\qquad\quad \mathit{Phase\_Range}\ (S_i, p_i, r) \wedge \\
&\qquad\quad \mathit{Phase\_Range}\ (S_j, p_j, r)
\end{aligned}$$

6.3.2. *Data Conflict Verification*

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, i.e.:

$$\vdash \quad \neg\ \mathit{Data\_Conflict} \quad \Leftrightarrow \quad \begin{aligned} &(\neg\ \mathit{RAW\_Conflict} \ \wedge \\ &\ \neg\ \mathit{WAR\_Conflict} \ \wedge \\ &\ \neg\ \mathit{WAW\_Conflict}) \end{aligned}$$

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:

$$\begin{aligned}
&\vdash \quad \forall\ I_1 \ldots I_{n_s} : \mathit{class\_instruction}. \\
&\qquad\qquad \neg\ \mathit{Multiple\_RAW\_Conflict}\ (I_1, \ldots, I_{n_s})
\end{aligned}$$

This goal includes a quantification over all possible conflict combinations that could occur between all permutations of n_s instructions within the pipeline. As in the case of resource conflicts, the direct proof of this goal results in a case explosion. Hence, we manage the proof in two steps as follows:

1. we first prove that dual conflicts do not occur:

$$\begin{aligned}
&\vdash \quad \forall\ I_i\, I_j : \mathit{class\_instruction}. \\
&\qquad \forall\ t_i^0\ t_j^0 : \mathit{Clock\_cycle}. \\
&\qquad\quad \neg\, \mathit{Dual\_RAW\_Conflict}\ ((I_i,\ t_i^0), (I_j,\ t_j^0))
\end{aligned}$$

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

$$\begin{aligned}
\vdash \quad & (\forall\, I_i\, I_j : \mathit{class\_instruction}\,.\\
& \quad \forall\, t_i^0\;\, t_j^0 : \mathit{Clock\_cycle}\,.\\
& \quad\quad \neg\, \mathit{Dual\_RAW\_Conflict}\,((I_i,\, t_i^0),\, (I_j,\, t_j^0)))\\
& \quad \Rightarrow (\forall\, I_1\, \ldots\, I_{n_s} : \mathit{class\_instruction}.\\
& \quad\quad\quad \neg\;\, \mathit{Multiple\_RAW\_Conflict}\,(I_1,\, \ldots,\, I_{n_s}))
\end{aligned}$$

The proof of step 2 is done in a straightforward manner since the universal quantification over all pairs (I_i, I_j) is more general than the disjunction over a fixed number of pairs depending on n_s. Using the definition of *Dual_RAW_Conflict* (cf. section 6.3.1), the goal for the first step is equivalent to:

$$\begin{aligned}
\vdash \quad & \forall\, I_i\, I_j : \mathit{class\_instruction}\,.\\
& \quad \forall\, t_i^0\;\, t_j^0 : \mathit{Clock\_cycle}\,.\\
& \quad\quad \forall\, r : \mathit{CL\_resource}\,.\\
& \quad\quad\quad \neg\;\, \mathit{Stage\_RAW\_Conflict}\,((I_i,\, t_i^0),\, (I_j,\, t_j^0),\, r)
\end{aligned}$$

The expansion of this goal at the stage level using the definition of *Stage_RAW_Conflict* yields either *True* or a number of subgoals, which include the specific resource and class instructions that conflict. 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 $\theta(s_i)$ - $\theta(s_j)$ = “$\theta(WB)$ - $\theta(ID)$”:

$$\begin{array}{l} (I_i = LOAD),\, (I_j = ALU),\;\; (0 < (t_j^0\, -\, t_i^0)) \\ (s_i = WB),\, (s_j = ID),\, (r = RF) \end{array} \;\vdash\;\; \neg\,((t_j^0\, -\, t_i^0) \leq 3)$$

This result is interpreted as follows: as long as the issue times of the conflicting LOAD and ALU-instructions satisfy the condition “$(t_j^0 - t_i^0) \leq 3$”, 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. “$(t_j^0 - t_i^0) = 3$”: with $t_i^u = (t_i^0 + \theta(WB))$ and $t_j^u = (t_j^0 + \theta(ID))$ (cf. section 6.3.1), this timing condition is equivalent to $((t_j^u - \theta(ID)) - (t_i^u - \theta(WB))) = 3$, i.e. $t_i^u = t_j^u$. Hence, referring to section 6.3.1, the data conflict should be explored at the lower time granularity of the phase level, by setting the following goal:

$$\vdash \neg\, \mathit{Phase\_RAW\_Conflict}\ (\mathit{ClassToStage\ (WB,\ LOAD),\ ClassToStage\ (ID,\ ALU),\ RF})$$

If the goal is proven correct, no data conflict happens, otherwise either the hardware EBM should be changed, e.g. via the inclusion of more clock phases, or one uses the software scheduling technique [43].

2. "$(t_j^0 - t_i^0) < 3$": 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**) since $(t_j^0 - t_i^0) < 3$ is equivalent to $(t_j^0 - t_i^0) \leq 2$.

Using the *software scheduling* technique (also called *instruction scheduling* [43]), we have to ensure that the issue time of a LOAD-instruction followed by an ALU-instruction should be at least 3 time units apart. For this example the given software constraint that leads to the proof of the dual data conflict goal, could then be defined as:

$$\begin{array}{l} \vdash_{def}\ SW\_Constraint := \\ \quad ((I_i = LOAD) \wedge (I_j = ALU) \wedge \\ \quad (r = RF) \wedge (0 < (t_j^0 - t_i^0)) \quad \Rightarrow \quad ((t_j^0 - t_i^0) > 3) \end{array}$$

Another widely used data conflict resolution technique is *bypassing* (also called *forwarding*) [43]. 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 behaviour is implemented in hardware by using some registers and corresponding feedback paths that hold and forward this data, respectively. Referring to the discussion in section 4.3, the implemented bypass behaviour should be specified in form of a predicate which ensures that by every data dependent instruction sequence the right data is forwarded to the EX-stage. For example, let *BYPASS_SPEC* be a predicate that describes the intended behaviour of the implemented hardware logic for data conflict resolution. This predicate specifies how the processor behaves in a case of a data conflict by detecting it and forwarding the right data to the right pipeline stage where it is needed. In order to prove that the hardware EBM implements this behaviour, we shall prove[8]:

$$\vdash\ EBM \Rightarrow BYPASS\_SPEC$$

Using the definition of *Stage_Range* and *Stage_Domain*, we easily extract from the predicate *BYPASS_SPEC* the following bypass condition:

$$\begin{array}{l} \vdash_{def}\ Bypass\_Cond := \\ \quad \forall\ I_i\ I_j : class\_instruction\ . \\ \qquad \exists\ rb\ .\ (rb = RF) \wedge Stage\_Range\ (I_i, EX, rb) \wedge Stage\_Domain\ (I_j, EX, rb) \end{array}$$

which formalizes the existence of the required buffers and bypass paths and thus we obtain:

$$\vdash\ BYPASS\_SPEC \Rightarrow Bypass\_Cond$$

Using transitivity we can derive:

$$\vdash\ EBM \Rightarrow Bypass\_Cond$$

Assuming this bypass condition 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 6.3.1) are set to EX and the timing condition is hence reduced to:

$$\ldots, (0 < (t_j^0 - t_i^0)) \ \vdash\ \neg\,((t_j^0 - t_i^0) \leq 0)$$

which is always true.

---

[8] A formal specification and verification of *BYPASS_SPEC* for a hardware implementation of DLX is beyond the scope of this paper and is reported elsewhere [27].

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

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

6.4. Control Conflicts

Control conflicts (also called *control hazards* [43, 67], *branch hazards* [43], *sequencing hazards* [57] or *branch dependencies* [35]) arise from the pipelining of branches and other instructions that change the program counter *PC*, i.e. 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 have been fetched before the program counter *PC* was updated. Thus straightforward program coding may yield incorrect results.

6.4.1. Control Conflict Specification

Let $A_f(I_i, t_i^0)$ be the fetch address of an instruction I_i issued at time t_i^0, i.e. $A_f(I_i, t_i^0) = PC(t_i^0)$, and let $A_n(I_i, t_i^0)$ be the address of the sequential next instruction (also called next address of I_i), i.e. $A_n(I_i, t_i^0) = PC(t_{i+1}^0)$. In a pipelined instruction execution, at each clock cycle a new instruction is issued (fetched), i.e. $PC(t_{i+1}^0) = PC(t_i^0+1)$. If I_i is a control instruction, then the sequential next address is a specific target address A_t, i.e. $A_n(I_i, t_i^0) = A_t(I_i, t_i^0)$. 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 $A_t(I_i, t_i^0)$ is equal to $PC(t_i^0+N)$, where $N > 1$. Hence, the next address is not equal to the target address, i.e.:

$$A_n(I_i, t_i^0) = PC(t_{i+1}^0) = PC(t_i^0+1) \neq PC(t_i^0+N) = A_t(I_i, t_i^0)$$

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 the resource *PC* that is not yet updated (written) by a previous instruction. This complies with the definition of RAW data conflict in *PC* [50] and thus the software control conflict could be defined as follows:

$$\vdash_{def} \mathit{Control\_Conflict} := \mathit{Stage\_RAW\_Conflict}\,((\mathit{CONTROL},\ t_i^0), (I_j,\ t_j^0), \mathit{PC})$$

6.4.2. Control Conflict Verification

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:

$$\begin{aligned} \vdash\ & \forall I_j : class\_instruction\ . \\ & \quad \forall t_i^0\ t_j^0 : Clock\_cycle\ . \\ & \qquad \neg\ Stage\_RAW\_Conflict\ ((CONTROL,\ t_i^0),\ (I_j,\ t_j^0),\ PC) \end{aligned}$$

and for the DLX processor example, we obtain, according to the four instruction classes, four subgoals of the following form:

$$(I_j = CLASS),\ (s_i = ID),\ (s_j = IF),\ (0 < (t_j^0 - t_i^0))\ \vdash\ \neg\ ((t_j^0 - t_i^0) \leq 1)$$

Since the issue times t_i^0 and t_j^0 satisfy $(0 < (t_j^0 - t_i^0))$, the timing condition for the control conflict $((t_j^0 - t_i^0) \leq 1)$ is equivalent to $((t_j^0 - t_i^0) = 1)$. Referring to the discussion on data conflict verification in section 6.3.2, we should check the conflict in this case at the phase level by setting the following goal (where IF_X represents the common IF-stage instruction of all instruction classes):

$$\vdash\ \neg\ Phase\_RAW\_Conflict\ (ID_C,\ IF_X,\ PC)$$

For the DLX example, we obtain:

$$(P_i = \phi_2),\ (P_j = \phi_1)\quad \vdash\quad \mathsf{F}$$

This result confirms the fact that the program counter *PC* (that should be the target address) can only be updated at the second clock phase of the ID-stage while it is needed for fetching in the first phase of IF.

For conflict resolution no bypassing is possible, since the calculation of the target address cannot be done earlier. One commonly used technique is software scheduling [43]. In the DLX RISC processor, we just need one delay slot $((t_j^0 - t_i^0) = 1)$ to ensure that control instructions are executed correctly. The given software constraint that is used in this case is defined as follows:

$$\vdash_{def}\ SW\_Constraint := ((I_i = CONTROL) \wedge (0 < (t_j^0 - t_i^0)) \Rightarrow ((t_j^0 - t_i^0) > 1)$$

Although delayed branching is used successfully for the reduction of the branch penalty[9], control conflicts could also be resolved in hardware using some special techniques, e.g. *branch prediction, branch folding*, etc. [30]. These techniques try to reach a nearly zero-delay branch, for example via the use a of branch history or a branch target buffer. However, there are different kinds of mechanisms that are implemented in different ways by different processors [54]. Hence, a general formalism cannot be given within the scope of our methodology. Referring to the discussion in section 4.3, the behaviour of the implemented branching mechanisms has to be specified formally and proven correct from the hardware implementation (as done for example in [46]). Using this formal specification of the resolution technique, the above timing condition for the avoidance of control conflicts has to be implied.

---

[9] Some leading statistics have shown that by 70% of the delayed branches, the first delay slot can be filled with a useful instruction, and by 25% the second one too [43].

To summarize, having used either an appropriate software constraint for a delayed branching or the conflict resolution technique in hardware (EBM), the non-existence of control conflicts is formally ensured, i.e.:

$$\boxed{\textit{EBM, SW\_Constraints} \vdash \neg\, \textit{Control\_Conflict}}$$

6.5. Summary

Our ultimate goal in proving the pipeline correctness relies in showing the non-existence of pipeline conflicts, i.e. resource, data and control conflicts. Given an adequate implementation EBM avoiding mutual resource use and involving conflict resolution mechanisms in hardware and/ or given some software constraints in form of timing conditions, we conclude from the theorems yielded in sections 6.2.2, 6.3.2 and 6.4.2:

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

and hence the pipeline correctness. The obtained proof of the conflicts freedom has been achieved at an abstract level by ranging over class instructions. Since neither structural, nor data or temporal abstraction exists between the architecture and class levels and consequently they involve the same resources and have the same timing behaviours, the obtained theorems for the pipeline correctness can be transferred to the architecture level. Hence, we have performed the proof for the pipeline correctness for all combinations of architectural instructions.

In contrast to the semantic correctness where, for a given RISC processor, a large number of verification goals is to be set and proven, the goals and proofs within the verification process for the pipeline correctness are fully processor independent. Further, with exception of the specification of the processor specific arguments of the enumeration types (cf. section 6.1), all required information about the specific processor that is to be verified are gained through mechanical extraction from the formal specifications of the already specified model levels (cf. section 3.2).

Furthermore, the verification method presented is constructive in that it helps the designer, within a post-design verification process, in validating some existing software or hardware constraints for conflict resolution or, within a verification-driven design process, in synthesizing the constraints needed for conflict resolution at a given step of the design process.

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 five automated proof tactics:

- one general tactic for deducing multiple (resource and data) conflicts at either the stage and phase levels,
- two tactics for verifying dual resource conflicts at the stage and phase levels, respectively, and
- two parameterized tactics for the verification of dual (RAW, WAR and WAW) data and control conflicts at the stage and phase levels, respectively.

Although we have been able to automate most of the verification process for pipeline conflicts using few parameterized tactics, manual steps still being necessary when undertaking the verification of data and control conflicts which are circumvented using processor specific hardware.

7. Implementation in HOL

All formal specification and proof strategies of our methodology have been implemented using the higher-order logic theorem prover HOL [34] (version HOL90.6 which is based on SML [61]) within the MEPHISTO verification framework [52]. The specification predicates for the model instructions, the hardware implementation description, the conflicts formalization, etc. are introduced in HOL as *definitions*. The goal setting functions are implemented as SML *functions*. The predicate extraction functions are implemented in HOL as *rules* which generate theorems form other theorems and definitions. The proof scripts (tactics) are implemented using SML functions and available HOL *tactics* and *tacticals* [34]. The implementation in HOL of the model specifications, the temporal abstraction function and the semantical correctness is reported in [71]. The HOL implementation of the pipeline conflict formalization and verification process is reported in [73].

All implementations are kept general so that it is applicable to a wide range of RISC processors and could be grouped as follows:

- implementations that need no instantiations and are directly useable for any RISC processor. These include the specification of pipeline conflicts (cf. sections 6.2.1, 6.3.1 and 6.4.1), the functions for predicate extractions (cf. section 6.1) and the proof tactics for pipeline verification (cf. section 6.5)
- implementations that have to be parameterized according to the handled RISC processor. These are the temporal abstraction function (cf. section 3.1), the goal setting functions and the proof tactics for the semantic verification (cf. sections 5.1, 5.2 and 5.3)
- implementations for which only general templates (illustrated by the DLX example) have been provided. These involve the specifications of the model levels (cf. section 3.2) and the type definitions for pipeline conflicts (cf. section 6.1)
- implementations for which no general patterns could be provided. These involve the specification and verification of specific hardware behaviours, e.g. interrupt, stalls, branch prediction, etc. (cf. section 4.3) which were mentioned within the scope of this paper through few pointers

Although the presented methodology has been implemented in HOL, its implementation within another verification system based on higher-order logic, e.g. Isabelle [62], PVS [59], Nuprl [24], SDVS [53], LAMBDA [2], etc. is also possible. The reason for our choice of HOL among the existing theorem provers is the fact that it has the largest support within the hardware verification community using theorem provers.

8. Experimental Results on a VLSI Implementation of DLX

The methodology presented so far, has been validated by using a VLSI implementation of DLX. The choice of the DLX architecture was motivated by the following facts:

- DLX includes the main features of existing RISC cores, such as Intel i860, MIPS R3000, Motorola M88000 and Sun SPARC
- existence of a well defined and thoroughly documented architectural description [43]
- frequent use of the DLX architecture as a benchmark example for different experimental purposes, e.g. performance analysis, simulation, verification, synthesis, etc.
- availability of already implemented variants of the DLX processor using different tools as VHDL [47] or GENESIL [56], e.g. [6, 10, 19, 40, 78]

This implementation of the DLX processor contains a five stage pipeline with a two phased clock, and its architecture includes 51 basic instructions (integer, logic, load/store and control). All these instructions are grouped into 5 classes according to which the stage and phase instructions are defined (in addition to the four classes in table 1, a fifth class for immediate ALU instructions has been provided). This architecture assumes synchronous instructions and data memories (caches) with an access time equal to one clock cycle. Further, all architectural (class) instructions are one cycle instructions, i.e. each clock cycle one instruction is completed and a new instruction is issued. Also, no branch prediction has been implemented and the branch technique provided is based on delayed branch with one delay slot. Hence, no pipeline stalls are necessary and therefore no stall mechanism was implemented. This DLX processor core has been designed and implemented within the commercial VLSI design environment CADENCE [17] using a 1.0 μm CMOS technology (figure 12). The implementation has approximately 150,000 transistors which occupy 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. A full description of the architecture and design of this DLX implementation is reported in [28].

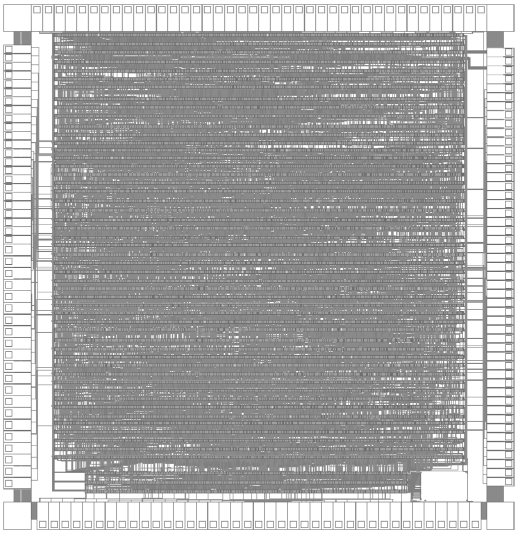

Figure 12. DLX VLSI Layout Picture

From the above given data, this DLX processor cannot be compared to commercial RISCs which include more than a million transistors. However, the core architecture of commercial processors usually do not contain a large number of transistors. For example, the core architecture of the i860 [48] represents only 30% of its 1.2 million transistors while the rest is used for cache, floating-point and other functional units [60]. Thus representing the complexity of 150,000 transistors for the DLX core architecture can be reckoned to be realistic enough. Considering the performance of the implemented DLX, its relatively low clock rate of 12.5 MHz is due to the fact that we used standard cells for our implementation while commercial processors use full-custom cells and a technology of less than 1.0 μm. Although the DLX processor is still simple when compared to commercial processors, its complexity is orders of magnitude greater than the complexities of reported verified processors as shown in table 2.

Table 2. Features of Reported Verified Processors

Processor Features	FM8501 [44]	VIPER [22]	Tamarack-3 [49]	Mini-Cayuga [66]	AVM-1 [77]	MTI [8]	DLX
Word Length	16-Bit	32-Bit	16-Bit	32-Bit	32-Bit	16-Bit	32-Bit
No. of Instructions	26	128	8	8	30	22	51
Microprogrammed	yes	no	yes	no	yes	yes	no
No. of Microinstructions	14	-	32	-	64	38	-
Pipelined	no	no	no	3-stage	no	no	5-stage
No. of Registers	16	4	2	32	32	32	32
Interrupt	no	no	yes	yes	yes	yes	yes
Memory Model	async.	sync.	async.	sync.	sync.	sync.	sync.
Memory Size	64 KB	1 MB	8 KB	1 GB	1 GB	8 MB	4 GB
Implemented	no	yes	no	no	no	yes	yes
Size (gates or transistors)	1,700 gt.	5,000 gt.	-	-	-	30,000 tr.	150,000 tr.

Using the already existing implementation of our methodology (cf. section 7), we have made the experiment by performing the verification of this DLX by a third person who implemented the processor in CADENCE. This person has an electrical engineering background with little knowledge in formal methods and without previous knowledge in HOL. He has been successful in specifying and verifying this DLX implementation within two months. However, most of this time was spent in learning about HOL, formally specifying the processor and verifying the processor specific interrupt and bypassing hardware. The following specifications have been provided:

- specification of the instructions of the architecture, class, stage and phase levels,
- formal description of the hardware implementation EBM down to the level of CADENCE standard cells,
- definition of the arguments for the instructions and pipeline types and
- specification of the interrupt and bypass behaviours.

For formal correctness the following verification tasks were involved:

- verification of the semantic correctness,
- verification of the pipeline correctness and
- verification of the interrupt and bypass behaviours.

With the exception of the bypass and interrupt behaviours the overall verification process has been achieved fully automatically. It is to be noted finally that during this experiment few bugs were found in the design which were not discovered during the simulation process. Examples of these bugs are the implementation of a wrong addressing of the register file at the WB-stage by immediate instructions and a missing bypass path for jump instructions that use the register file during the ID-stage. The first failure arose during the semantic (class level) correctness proof and the second one during the pipeline (data conflict) verification. Due to the hierarchical and constructive aspects of our methodology, these bugs were easily fixed and recovered.

All formal specifications and verification proofs have been done within the HOL verification system (version HOL90.6) on a SPARC10 with a 128 MB main memory. The specification overhead, the run times and the number of created inferences for the verification of this DLX processor example are given in detail in the tables to follow. The overall specification for the DLX core (illustrated in table 3 via the code length in number of lines and by the file size in Bytes) is about 4500 lines long of which about 70% corresponds to the description of the EBM. The run times (including the time for goal setting) for the proofs of the semantic correctness of the whole processor are given in table 4. Hereby it is interesting to notice that, as expected, the verification of the phase level corresponds to about 90% of the total semantic correctness proof overhead. The run times for the theorem generation of the *Used* and *Range/Domain* predicates are given in table 5. The run times for the pipeline verification for the implemented DLX processor are given in table 6. The overall proof results including the verification of the interrupt and bypass behaviours are summarized in table 7. Accordingly, the whole verification of this DLX implementation took about one hour and required about seven millions inferences.

Table 3. Formal Specifications

Specification	**# Lines**	**# Bytes**	**Comments**
Architecture Level	718	27710	51 instructions
Class Level	216	8737	5 instructions
Stage Level	219	7845	13 instructions
Phase Level.	226	7149	26 instructions
EBM	3144	123121	-
INTERRUPT_SPEC	64	1700	-
BYPASS_SPEC	50	2211	-
Type Definitions	78	3610	-
Σ Specification	4515	182083	-

Table 4. Semantic Correctness

Verification Goal	**Time (in sec)**	**# Inferences**	**Comments**
Stage Level ⇒ Class Level	27.34	34640	5 theorems
Phase Level ⇒ Stage Level	23.43	22074	13 theorems
EBM ⇒ Phase Level	850.48	204521	26 theorems
Architecture Level (instantiations)	14.08	5719	51 theorems
Σ Semantic Correctness	915.33	266945	-

Table 5. Predicates Extractions

Predicate	**Time (in sec)**	**# Inferences**	**Comments**
Stage_Used	206.74	576515	180 theorems generated
Phase_Used	1087.05	2536492	360 theorems generated
Stage_Range/Domain	302.00	534283	250 theorems generated
Phase_Range/Domain	266.70	267926	260 theorems generated
Σ Predicates Extractions	1862.49	3915216	-

Table 6. Pipeline Correctness

Verification Goal	**Time (in sec)**	**# Inferences**	**Comments**
Resource Conflicts	674.81	1455146	0 conflicts
RAW Data Conflict	536.86	1787020	15 conflict cases (3 slots) by RF and 5 conflict cases (1 slot) by PC
(using SW-Scheduling)	294.05	159806	0 conflicts
(using Bypassing)	1.89	5438	0 conflicts
WAR Data Conflict	578.07	1749153	0 conflicts
WAW Data Conflict	576.55	1735142	0 conflicts
Control Conflict	36.00	27659	5 conflict cases (1 slot)
(using SW-Scheduling)	47.70	34020	0 conflicts
Σ Pipeline Correctness	2402.29	6754120	-

Table 7. Summary of the Verification Results

Verification Goal	**Time (in sec)**	**# Inferences**	**Comments**
EBM ⇒ BYPASS_SPEC	24.29	2706	proof done manually
EBM ⇒ INTERRUPT_SPEC	636.11	20725	proof done manually
Semantic Correctness	915.33	266945	95 main theorems
Pipeline Correctness	2402.29	6754120	3 main theorems
Σ DLX Verification	3978.02	7044496	-

9. Conclusions

In this paper we have shown the feasibility of formal verification techniques when applied cleverly to specific classes of circuits. In this sense, we have provided a practical methodology for the formal verification of RISC processor cores. This methodology is based on a novel hierarchical interpreter model which is applicable for RISC cores in general. This model is a modification of the one given by Anceau [3] for designing microprogrammed processors and reflects the design hierarchy which is used for designing real pipelined RISC processors. Hence, the methodology based on it can be used by computer architecture designers for successively refining and verifying their designs. Further, the hierarchy present in the model can be exploited, to split the overall verification task into a number of manageable subtasks so that the designers can formally verify their designs during the design phase itself.

Due to the parallelism in the execution of instructions resulting from the pipelined architecture of RISCs, a meticulous temporal abstraction has been developed and implemented. The correctness of the RISC processor is ensured by splitting the proof goal into two independent parts, namely the correct implementation of the semantics of each single instruction and the correctness of the pipelined execution of various instructions by the hardware. The ease of formalizing the specifications in higher-order logic at each level of abstraction and the similarity of the proofs between the levels have lead to general functions and proof tactics which automate the goal setting and the correctness proof for the semantic correctness. Furthermore, 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 and verified using few parameterized proof scripts. 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. Furthermore, within the verification of the pipeline correctness, we have adapted constructive proofs for conflicts verification 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 were kept general and provide a pattern to follow when verifying RISC cores. The specification and verification templates give which definitions must be specified and which goals must be proved to verify the machine. Given such specifications and a description of the hardware implementation, the proof process has been automated by using parametrizable tactics. These tactics are independent of the underlying implementation and can be used for a large number of RISC cores. The whole methodology is generic, in that it is applicable to RISC cores with any pipeline depth and hence to superpipelined architectures.

While exercising the verification process, we discovered that the proofs can be hierarchically managed in a top-down or bottom-up manner, so that a verification-driven design or a post-design verification can be performed. Within the scope of this paper, the correctness proofs were mainly handled by means of the top-down verification-driven design methodology. By the application of the methodology on the implemented DLX processor, for example, we have handled the verification of the pipeline correctness in a post-design manner (as described in [72]), in that the verification was done more or less in a single step through all hierarchies [27].

We have implemented the different specifications and proof strategies at each level of abstraction in HOL within the MEPHISTO verification framework which is linked to the commercial VLSI design tool CADENCE. The entire methodology has been validated by using a VLSI implementation of DLX. The feasibility of the verification techniques developed is illustrated by the run times reached for the verification of this realistic RISC core. In our future work, we shall extend the layer of the core to superscalar architectures including pipelined functional units, multiple instruction issue, etc.

References

1. M. Aagaard; M. Leeser: *Reasoning about Pipelines with Structural Hazards*; Proc. Theorem Provers in Circuit Design, Bad Herrenalb, Germany, September 1994, pp. 15-34.
2. Abstract Hardware Limited: *LAMBDA — Logic and Mathematics behind Design Automation*; User and Reference Manuals, Version 3.1, 1990.
3. F. Anceau: *The Architecture of Microprocessors*; Addison-Wesley Publishing Company, 1986.
4. P. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth though Proof*; Academic Press, 1986.
5. T. Arora: *The Formal Verification of the VIPER Microprocessor: EBM to Phase, Phase to Microcode Level*; Master's thesis, University of California, Davis, 1990.
6. P. Ashenden: *DLX VHDL Model*; Department of Computer Science, University of Adelaide, Australia, November 1993.
7. T. Baker: *Headroom and Legroom in the 80960 Architecture*; Proc. 35th IEEE Computer Society International Conference (COMPCON90), San Francisco, California, February 1990, pp. 299-306.
8. D. Borrione; P. Camurati; J. Paillet; P. Prinetto: *A Functional Approach to Formal Hardware Verification: The MTI experience*; Proc. IEEE International Conference on Computer Design (ICCD88), Rye Brook, New York, October 1988, IEEE Computer Society Press, pp. 592-595.
9. V. Bhagwati; S. Devadas: *Automatic Verification of Pipelined Microprocessors*; Proc. ACM/IEEE 31st Design Automation Conference (DAC94), San Diego, California, June1994, pp. 603-608.
10. M. Blomkvist; J. Nilsson; W. Sagefalk: *A VLSI Implementation of the DLX Microprocessor*; Department of Computer Engineering, Lund University, Sweden, September 1992.

11. S. Bose; A. Fisher: *Verifying Pipelined Hardware using Symbolic Logic Simulation*; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, Massachusetts, September 1989, IEEE Computer Society Press, pp. 217-221.
12. A. Bode: *RISC-Architekturen*; BI-Wiss. Verlag, 1990.
13. R. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*; IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986, pp. 677-691.
14. A. Bronstein; C. Talcott: *Formal Verification of Pipelines based on String-Functional Semantics*; In: L. Claesen (Ed.), Formal VLSI Correctness Verification, VLSI Design Methods II, Elsevier Science Publishers B. V. (North-Holland), 1990, pp. 349-367.
15. O. Buckow: *Formale Spezifikation und (Teil-) Verifikation eines SPARC-kompatiblen Prozessors mit LAMBDA*; Diplomarbeit, Fachbereich Mathematik-Informatik, Universität-Gesamthochschule Paderborn, Germany, October 1992.
16. J. Burch; D. Dill: *Automatic Verification of Pipelined Microprocessor Control*; In: D. Dill (Ed.), Computer Aided Verification, Lecture Notes in Computer Science 818, Springer Verlag, 1994, pp. 68-80.
17. Cadence Design Systems Inc.: *CADENCE User Manuals*; Cadence Design Systems Inc., October 1991.
18. A. Camilleri: *Simulation as an Aid to Verification Using the HOL Theorem Prover*; Technical Report No. 150, Computer Laboratory, Cambridge University, October 1988.
19. CAO-VLSI Team: *Implementation of DLX in ALLIANCE*; MASI Laboratory, University Pierre et Marie Curie, Jussieu, Paris, France, March 1993.
20. P. Camurati; P. Prinetto: *Formal Verification of Hardware Correctness: Introduction and Survey of Current Research*; IEEE Computer, July 1988, pp. 8-19.
21. R. Cloutier; D. Thomas: *Synthesis of Pipelined Instruction Set Processors*; Proc. ACM/IEEE 30th Design Automation Conference (DAC93), Dallas, Texas, June 1993, pp. 583-588.
22. A. Cohn: *A Proof of the Viper Microprocessor: The First Level*; In: G. Birtwistle and P. Subrahmanyam (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988.
23. A. Cohn: *The Notion of Proof in Hardware Verification*; Journal of Automated Reasoning, Vol. 5, 1989, pp. 127-139.
24. R. Constable et al.: *Implementing Mathematics with the Nuprl Proof Development System*; Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
25. J. Cook: *Verification of the C/30 Microcode Using the State Delta Verification System (SDVS)*; Proc. 13th National Computer Security Conference, Washington, D.C., National Bureau of Standards/National Computer Security Centre, October 1990, pp. 20-31.
26. D. Cyrluk: *Microprocessor Verification in PVS: A Methodology and Simple Example*; Technical Report SRI-CSL-92-12, SRI Computer Science Laboratory, December 1993.
27. M. Dehof: *Formale Spezifikation und Verifikation des DLX-RISC-Prozessors*; Diplomarbeit, Institut für Technik der Informationsverarbeitung, Universität Karlsruhe, Germany, August 1994.
28. M. Dehof; S. Tahar: *Implementierung des DLX RISC-Processors in einer Standardzellen-Entwufsumgebung*; Technical Report No. SFB 358-C2-1/94, Institute for Computer Design and Fault Tolerance, University of Karlsruhe, Germany, March 1994.
29. Digital Equipment Corp.: *Alpha Architecture Handbook*; Digital Equipment Corp., Maynard, Massachusetts, Order No. EC-H1689-10, 1992.
30. P. Dubey; M. Flynn: *Branch Strategies: Modelling and Optimization*; IEEE Transactions on Computer, Vol. 40, No. 10, October 1991, p. 1159-1167.
31. Electronic Design Interchange Format, *Version 2 0 0: EIA Interim Standard No. 44*; EDIF Steering Committee, Electronic Industries Association, May 1987.
32. S. Furber: *VLSI RISC Architecture and Organization*; Electrical Engineering and Electronics, Dekker, New York, 1989.
33. G. Gopalakrishnan; R. Fujimoto; V. Akella; N. Mani; K. Smith: *Specification-Driven Design of Custom Hardware in HOP*; In: G. Birtwistle and P. Subrahmanyam (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1989, pp. 128-170.
34. M. Gordon; T. Melham: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*; Cambridge, University Press, 1993.
35. A. Van De Goor: *Computer Architecture and Design*; Addison-Wesley, 1989.
36. G. Gopalakrishnan: *Specification and Verification of Pipelined Hardware in HOP*; In: J. Darringer and J. Rammig (Eds.), Computer Hardware Description Language and their Applications (CHDL89), Elsevier Science Publishers B.V. (North-Holland), 1989, pp. 117-131.
37. M. Gordon: *Proving a Computer Correct using the LCF_LSM Hardware Verification System*; Technical Report No. 42, Computer Laboratory, University of Cambridge, September 1983.
38. B. Graham: *The SECD Microprocessor: A Verification Case Study*; Kluwer Academic Publishers, 1992.

39. A. Gupta: *Formal Hardware Verification Methods: A Survey*; Journal of Formal Methods in System Design, Vol. 1, No. 2/3, 1992, pp. 151-238.
40. A. Gupta; P. Stephan: *VHDL Design and Analysis of DLX*; CS252 Semester Project, University of California at Berkeley, May 1991.
41. 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.
42. F. Hanna; M. Longley; N. Daeche: *Formal Synthesis of Digital Systems*; In: L. Claesen (Ed.), Applied Formal Methods for Correct VLSI Design, Elsevier Science Publishers B. V. (North-Holland), 1989, pp. 532-548.
43. J. Hennessy; D. Patterson: *Computer Architecture: A Quantitative Approach*; Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
44. W. Hunt: *The Mechanical Verification of a Microprocessor Design*; In: D. Borrione (Ed.), From HDL Description to Guaranteed Correct Circuit Designs, Elsevier Science Publishers B.V. (North-Holland), 1987, pp. 89-129.
45. W. Hunt: *Microprocessor Design Verification*; Journal of Automated Reasoning, Vol. 5, No. 4, 1989, pp. 429-460.
46. W. Hwu; P. Chang: *Efficient Instruction Sequencing with Inline Target Insertion*; IEEE Transactions on Computer, Vol. 41, No. 12, December 1992, pp. 1537-1551.
47. Institute of Electrical and Electronics Engineers: *IEEE Standard VHDL Language Reference Manual*; IEEE Press, New York, June 1993.
48. Intel Corporation: *i860 64-Bit Microprocessor Programmer's Reference Manual*; Intel Corporation, Santa Clara, California, 1989.
49. J. Joyce: *Multi-Level Verification of Microprocessor-Based Systems*; PhD. Thesis, Computer Laboratory, Cambridge University, December 1989.
50. P. Kogge: *The Architecture of Pipelined Computers*; McGraw-Hill, 1981.
51. T. Kropf; R. Kumar; K. Schneider: *Embedding Hardware Verification within a Commercial Design Framework*; Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 93), Lecture Notes in Computer Science, Springer Verlag, 1993.
52. R. Kumar; K. Schneider; T. Kropf: *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.
53. L. Marcus: *SDVS 10 Users' Manual*; Technical Report ATR-91(6778)-10, The Aerospace Corporation, 1991.
54. S. McFarling; J. Hennessy: *Reducing The Cost of Branches*; Proc. 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 1986.
55. T. Melham: *Abstraction Mechanisms for Hardware Verification*; In: G. Birtwistle and P. Subrahmanyam, (Eds.), VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1988, pp. 129-157.
56. Mentor Graphics Inc.: *GENESIL Designer Manuals*; Mentor Graphics Inc., September 1989.
57. V. Milutinovic: *High Level Language Computer Architecture*; Computer Science Press, Inc., 1989.
58. Motorola, Inc.: MC88100 RISC Microprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall, 1988.
59. S. Owre; N. Shankar; J. Rushby: *User Guide for the PVS Specification and Verification System, Language, and Proof Checker*; Computer Science Laboratory, SRI International, Melno Park, California, February 1993.
60. P. Patel; D. Douglass: *Architecture Feature of the i860 - Microprocessor RISC Core and on-Chip Caches*; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, MA, September 1989, IEEE Computer Society Press, pp. 385-390.
61. L. Paulson: *ML for the Working Programmer*; Cambridge University Press, 1991.
62. L. Paulson: Isabelle: *A Generic Theorem Prover*; Lecture Notes in Computer Science 828, Springer Verlag, 1994.
63. A. Roscoe: *Occam in the Specification and Verification of Microprocessors*; Philosophical Transactions of the Royal Society of London, Series A: Physical Sciences and Engineering, Vol. 339, No. 1652, April 1992, pp. 137-151.
64. R. Sekar; M. Srivas: *Formal Verification of a Microprocessor Using Equational Techniques*; In: G. Birtwistle and P. Subrahmanyam (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer Verlag, 1989, pp. 171- 217.
65. J. Saxe; S. Garland; J. Guttag; J. Horning: *Using Transformations and Verification in Circuit Design*; Proc. 2nd Workshop on Designing Correct Circuits, Lyngby, Danmark, January 1992.
66. M. Srivas; M. Bickford: *Formal Verification of a Pipelined Microprocessor*; IEEE Software, Vol. 7, No.5, September 1990, pp. 52-64.
67. H. Stone: *High-Performance Computer Architecture*; Addison-Wesley Publishing Company, 1990.
68. Sun Microsystems, Inc.: *The SPARC Architecture Manual*; Sun Microsystems, Inc., USA, Version 8, Part No. 800-1399-09, August 1989.

69. E. Talkhan; A. Ahmed; A. Salama: *Microprocessors Functional Testing*; IEEE Transactions on Computer Aided Design, Vol. 8, No. 3, March 1989.
70. S. Tahar; R. Kumar: *Towards a Methodology for the Formal Hierarchical Verification of RISC Processors*; Proc. IEEE International Conference on Computer Design (ICCD93), Cambridge, Massachusetts, October 1993, IEEE Computer Society Press, pp. 58-62.
71. S. Tahar; R. Kumar: *Implementing a Methodology for Formally Verifying RISC Processors in HOL*; In: J. Joyce and C. Seger (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 281-294.
72. S. Tahar; R. Kumar: *Formal Verification of Pipeline Conflicts in RISC Processors*; Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September 1994, IEEE Computer Society Press, pp. 285-289.
73. S. Tahar; R. Kumar: *Implementational Issues for Verifying RISC-Pipeline Conflicts in HOL*; In: T. Melham and J. Camilleri (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 854, Springer Verlag, 1994, pp. 424-439.
74. M. Thomas: *The Industrial Use of Formal Methods*; Microprocessor and Microsystems, Vol. 17, No. 1, 1993, pp. 31-36.
75. N. Tredemick: *Experiences in Commercial VLSI Microprocessor Design*; Microprocessors and Microsystems, Vol. 12, No.8, October 1988.
76. P. Villarrubia; Nusbaum, G.; Masleid, R.; Patel, P.: *IBM RISC Chip Design Methodology*; Proc. IEEE International Conference on Computer Design (ICCD89), Cambridge, Massachusetts, September 1989, IEEE Computer Society Press, pp. 143-147.
77. P. Windley: *The Formal Verification of Generic Interpreters*; PhD. Thesis, Division of Computer Science, University of California, Davis, July 1990.
78. K. Winters: *ASIC Design Experience: MDLX*; Department of Electrical Engineering, Montana State University, USA, April 1992.
79. W. Wong: *Modelling Bit Vectors in HOL: the word Library*; In: J. Joyce and C. Seger (Eds.), Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science 780, Springer Verlag, 1994, pp. 371-384.