A Practical Methodology for the Formal Verification of RISC Processors

SOFIÈNE TAHAR tahar@ece.concordia.ca
Dept. of Electrical & Computer Engineering, Concordia University, Montreal, Canada

RAMAYYA KUMAR kumar@verysys.com
Verysys GmbH, Berlin, Germany

Abstract. In this paper a practical methodology for formally verifying RISC cores is presented. Using a hierarchical model which reflects the abstraction levels used by designers of real RISC processors, proofs between neighboring levels are performed for simplifying the verification process. The proofs are performed by showing that each instruction is executed correctly by the pipelined machine with respect to the semantics of the instruction set architecture. During this proof, temporal abstractions are used to find correspondences between the various levels of abstractions. Additionally, lower level implementational details such as, multi-phased clocks and gate level descriptions of the final implementation, are accounted for. The overall correctness proof is managed in two complementary steps, namely, *pipeline data* and *pipeline control* correctness. In the former, we show that the cumulative effect of pipeline suboperations yields the data semantics of architecture instructions. While in the latter, we are concerned with interferences (conflicts) between the different instructions and suboperations in the pipeline. We have developed a set of parametrized proof scripts which highly automate the different proof tasks. In addition, the pipeline control proof is constructive, in the sense that the conditions under which the pipeline conflicts occur are automatically generated and explicitly stated thus aiding the user in its removal. All developed specifications and proof scripts are kept general, so that the methodology could be applied for a wide range of RISC cores (e.g. those used in embedded systems). 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 [23]. 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 [79]. 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* [43] is an alternative approach that systematically analyses all cases in a design and specification [27].

In the recent past several successful microprocessor specification and verification efforts have been performed using formal methods; some using high-order logic [26, 41, 42, 53, 71, 83] and others based on functional calculi [10, 29, 48, 68, 70]. Among the processors verified within these works only the VIPER [26], the C/30 [29] and the AAMP5 [71] processors are commercial ones, however, their verification was only partly achieved. With exception of these three 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 [83], 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, we choose this powerful logic as underlying formalism for our work. But, since this logic is neither complete nor decidable [4], no automated proofs can be provided in general. To circumvent this disadvantage, we propose in this work the development of appropriate heuristics and techniques which automate the verification for 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 and typical steps in verification, which can be simultaneously exploited to provide automation.

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 [14]. However, additional problems such as dealing with 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 multicycles 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 endeavor 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.

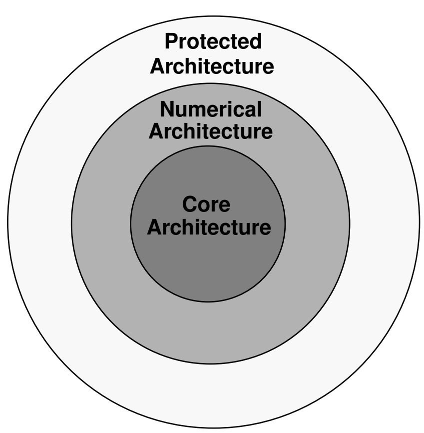

Figure 1. Multiple Layered RISC Architecture

Recently, there have been successful efforts for verifying pipelined processors using theorem-provers [1, 17, 30, 67, 69, 70, 71, 82, 84]. However, in all these cases, either the processor was extremely simple (e.g. in [30, 69] a very simple 3-stage pipeline known as Saxe-pipeline is handled without taking the actual implementation into account) or a large amount of labor was required. Among these works, only [17] deals with the verification of a real RISC processor, namely a SPARC model [73]. Still, this work was only able to verify parts of the processor at certain levels of abstraction. Besides this SPARC model verification, the work in [24, 82] also handles the verification of a RISC processor called Sawtooth which is comparable to the DLX RISC processor investigated in this paper. Several techniques used in [24, 82] are similar to the techniques presented in the current paper, however, the authors explicitly make extensive reference to earlier publications [75, 76, 77, 78] of our work on this subject.

Lately, automated techniques for the verification of pipelined processors have been presented [11, 18, 19]. However, due to the computational cost of BDD manipulations [15], the method presented in [11] 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 those described in [18, 19] deal with the verification of the control part, and additionally abstract the behavior of the datapath components. Moreover, these efforts do not reflect the overall design aspects of real RISC processors (e.g. only few instructions, restriction to an abstract model of the hardware, no consideration of clock phase granularities, no interrupts, etc.) and also involve intelligent *manual* interactions before the automatic process can proceed. Recently, results have been reported for checking the correctness of these manual steps via a theorem prover [85], since the abstractions to be performed are higher-order in nature and moreover problem-dependent. 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 [13, 16, 40]. A short survey of the different approaches is given in [84]. In contrast to all related works, we are developing a methodology and an associated environment for the routine verification of the major properties of RISC cores. We are investigating the verification of RISC designs from the specification of instruction sets down to their circuit implementations, independent of the data width and including features of real RISCs as behavior at the clock phase granularity, hierarchical implementation of the hardware using commercial cell libraries, interrupts handling, 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 [38] and to integrate it into a general verification framework MEPHISTO [56]

The organization of this paper is as follows: Section 2 describes the DLX processor which will be used as a vehicle for the illustration of the methodology. Section 3 proposes a novel hierarchical model for RISCs on which the verification process will be based. Section 4 gives an overview of the verification tasks. Section 5 sketches a new temporal abstraction mechanism and section 6 describes a formal specification of the model levels. The overview of the verification tasks, given in section 4, will be explored in detail in the sections 7, 8 and 9. Section 10 briefly describes several aspects of the implementation of the presented methodology in HOL. Section 11 contains some experimental results based on the verification of a VLSI implementation of the DLX RISC processor and section 12 finally concludes the paper.

2. DLX Description

Throughout the paper, most of the methods and techniques presented will be exercised, for illustration purposes, by means of a VLSI implementation of a RISC example — DLX [47]. 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 of an hypothetical DLX [47]
- 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 (e.g. [6, 12, 22, 44, 86]) using different tools as VHDL [51] or GENESIL [60].

The implementation of the DLX processor used in this paper contains a five stage pipeline with a two phased clock. Its architecture includes 51 basic instructions grouped in 4 classes (integer, logic, load/store and control) using three instruction formats. This DLX has a 32-bit load/store architecture with harvard memory configuration, i.e., separate instruction and data memories (caches), and includes 32 multipurpose registers. The architecture assumes synchronous instruction and data memories (caches) with an access time equal to one clock cycle. Hence, all instructions (including loads and stores) are single cycle instructions, i.e. in each clock cycle one instruction is completed and a new instruction is issued. For pipeline conflicts resolution, this DLX provides a bypassing logic hardware. Branch conflicts are resolved using delayed branch based on software scheduling with one delay slot. The hardware does not provide any interlock mechanism for supporting pipeline stalls, hence relying on bypassing and software scheduling to resolve pipeline conflicts. This design decision actually optimizes the performance of the current DLX since the penalty by conflicts is in worst case one clock cycle. Therefore, no expensive dynamic branch prediction or advanced dynamic scheduling techniques (as scoreboarding or reservation stations) have been implemented. Furthermore, the implemented DLX provides circuitry for handling precise interrupts (exceptions). Four priority levels of interrupts (illegal opcode, overflow, memory misalignment and external interrupts) are supported. Finally, this processor does not integrate on-chip any floating-point unit (FPU), cache or memory management unit (MMU).

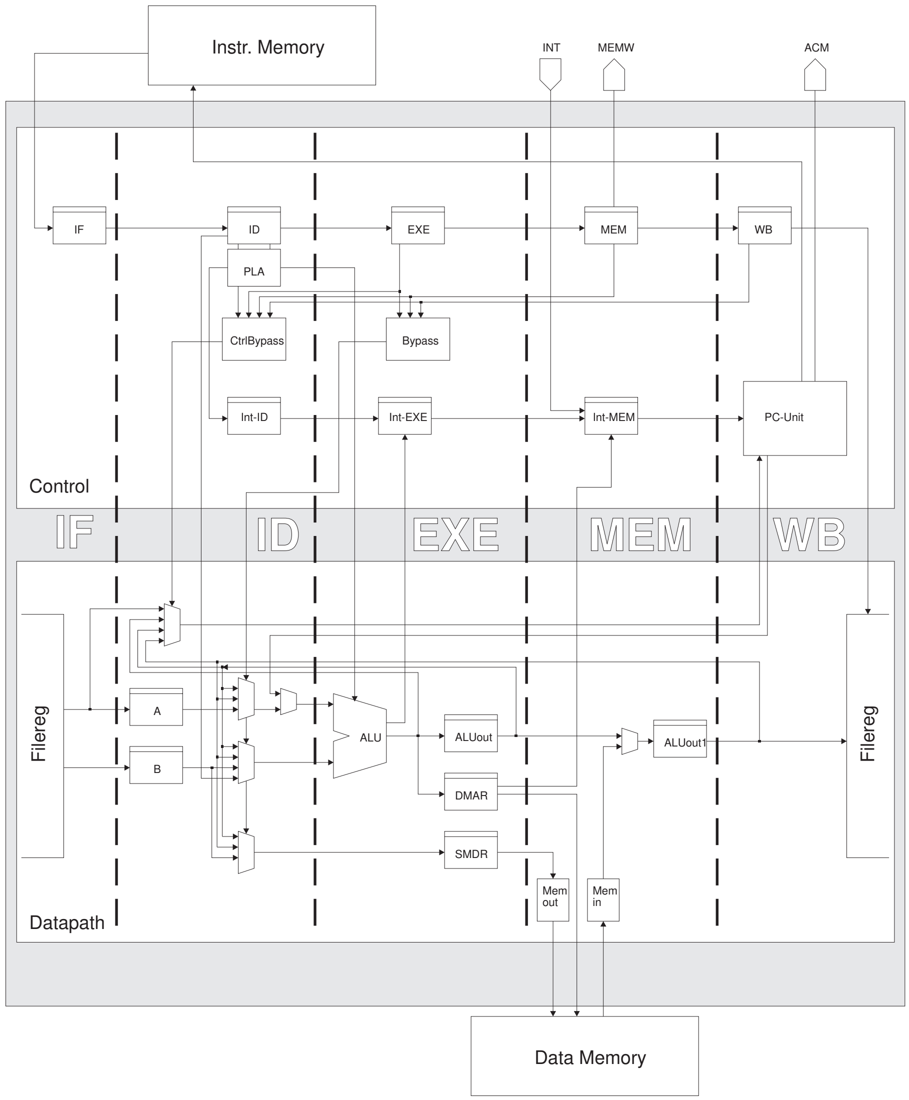

Figure 2. Hardware Implementation of DLX (simplified)

Figure 2 shows a simplified form of the hardware of the DLX processor implementation [32]. The datapath and the control unit implement the pipelined execution. They are physically composed of a series of functional units and pipeline buffers 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.

This DLX processor has been designed and implemented within the commercial VLSI design environment CADENCE [20] using a 1.0 µm CMOS technology (figure 3). 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 runs at a clock rate of 12.5 MHz. A full description of the architecture and design of this DLX implementation is reported in [32].

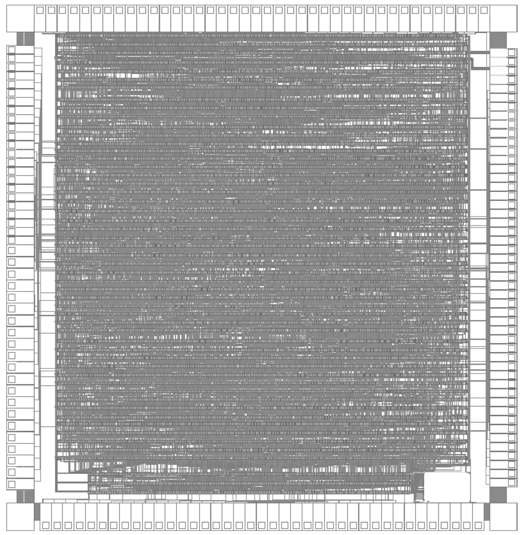

Figure 3. Layout of DLX VLSI Implementation

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 [52] represents only 30% of its 1.2 million transistors while the rest is used for cache, floating-point and other functional units [64]. Thus the complexity of 150,000 transistors for the DLX core architecture can be reckoned to be realistic enough, especially if such a core is used in embedded processors. Considering the performance of the implemented DLX, its relatively low clock rate of 12.5 MHz is due to the usage of standard cells for our implementation as opposed to the usage of full-custom cells and better technologies in commercial processors. 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 1.

Table 1. Features of Reported Verified Processors

Processor / **Features**	FM8501 [48]	VIPER [26]	Tamarack-3 [53]	Mini-Cayuga [70]	AVM-1 [83]	MTI [10]	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.

Table 2, for instance, shows the pipeline structure of DLX [47] with four example instructions ADD (addition), LW (load word), SW (store word) and JMP (jump). This pipeline structure lists the sets 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 instructions, respectively. There are five pipeline stages IF (instruction fetch), ID (instruction decode), EX (instruction execute), MEM (memory access), and WB (result write-back) and two clock phases ϕ_1 and ϕ_2. In table 2, "$\leftarrow$" means that the stage transfer is not broken down into phase transfers. "$\underset{\phi_1}{\leftarrow}$", "$\underset{\phi_2}{\leftarrow}$" represent transfers taking place in phase 1 or 2, respectively.

Table 2. DLX Pipeline Structure.

	ADD	**LW**	**SW**	**JMP**
IF	IR $\leftarrow$ I-MEM [PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ I-MEM [PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ I-MEM [PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ I-MEM [PC] PC $\leftarrow$ PC+4
ID	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 $\leftarrow$ IR	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 $\leftarrow$ IR	A $\underset{\phi_2}{\leftarrow}$ RF[rs1] B $\underset{\phi_2}{\leftarrow}$ RF[rs2] IR1 $\leftarrow$ IR	BTA $\underset{\phi_1}{\leftarrow}$ PC + (IR) PC $\underset{\phi_2}{\leftarrow}$ BTA
EX	ALUOUT $\leftarrow$ A + B	DMAR $\leftarrow$ A + (IR1)	DMAR $\leftarrow$ A+(IR1) SMDR $\leftarrow$ B	
MEM	ALUOUT1 $\leftarrow$ ALUOUT	LMDR $\leftarrow$ D-MEM[DMAR]	D-MEM[DMAR] $\leftarrow$ SMDR	
WB	RF[rd] $\underset{\phi_1}{\leftarrow}$ ALUOUT1	RF[rd] $\underset{\phi_1}{\leftarrow}$ LMDR		

Following storage elements are used:

- IR, IR1: instruction registers
- PC: program counter
- I-MEM: instruction memory
- D-MEM: data memory
- RF: register file
- A, B: ALU operands registers
- ALUOUT, ALUOUT1: ALU result registers
- BTA: branch target address register
- DMAR: data memory address register
- SMDR: store memory data register
- LMDR: load memory data register

3. 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 [49, 53, 82, 83]. 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 behavioral 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 neighboring levels is reduced.

3.1. CISC Interpreter Model

In some related work an interpreter model for microprogrammed processors has been presented [5, 53, 83]. This CISC interpreter model is given in figure 4 (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 the instruction executions. 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.

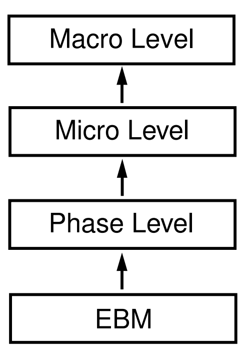

Figure 4. CISC Interpreter Model

The way RISC designs are carried out and structured is different from that of CISCs, e.g. the micro level does no more exist because of the hardwired control, also pipelining is an integrated concept of RISCs, etc. The mentioned structuring of the specification using the CISC interpreter model is hence unsuitable for RISC cores [75] and we have to look for another verification model.

3.2. 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 those register transfers that occur during a specific 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 [39]. 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 [26], 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] [75]. 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 RISC processor [33, 52, 62, 73]. Furthermore, a group of instructions belonging to one class exhibit the following properties: use the same number of pipeline stages, are executed by the same stage instructions, and are 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

1. This Notion of instruction classes is being also adopted by RISC designers in order to improve the pipeline execution [47] as well as for simulation [81], synthesis [25] or testing purposes [74].

(e.g. jumps, branches, etc.), they are implemented in only two pipeline stages. 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 operations of realistic RISCs, it can be seen that not all stage operations are broken down into phases [47]. Such phase level instructions can be modeled 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 3 shows the class abstracted pipeline structure of DLX [47] 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 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 correspond to the first letter of each 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 3. In some cases, we will use the subscript "*X*" to denote a stage or phase instruction that is shared by all class instructions, e.g. IF_X represents the common IF-stage instruction of all instruction classes.

Table 3. Class Abstracted 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	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$A \xleftarrow[\phi_2]{} RF[rs1]$ $B \xleftarrow[\phi_2]{} RF[rs2]$ IR1 ← IR	$BTA \xleftarrow[\phi_1]{} \mathbf{f_C}(PC, IR, RF)$ $PC \xleftarrow[\phi_2]{} BTA$
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	$RF[rd] \xleftarrow[\phi_1]{} ALUOUT1$	$RF[rd] \xleftarrow[\phi_1]{} LMDR$		

Due to the introduction of instruction classes, abstraction functions (f_A, f_L, f_S and f_C) replace the actual architectural instructions and are used in the relevant stage instructions for each class[2]. In comparison to table 2 which defined the pipeline structure for four architectural instructions,

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

namely, ADD, LW, SW and JMP, table 3 defines the pipeline structure for all instruction classes of the DLX. 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 5. 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 behavior and will be specified independently. Namely, from the instruction set of the RISC core, we extract the formal specification of the architecture and class levels, from the pipeline architecture, we derive the specification of the stage and phase levels, and from the current hardware implementation, we formally describe the design structure EBM.

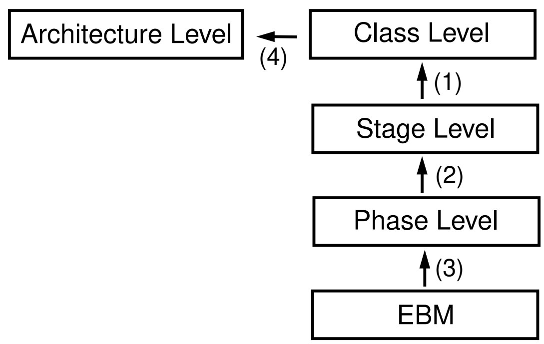

Figure 5. RISC Interpreter Model

Using the steps (1), (2) and (3), as shown in figure 5, 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 hardware implementation EBM and the architectural 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 behavioral descriptions [27]. 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 the actual architecture by replacing the class abstraction function by concrete operations. Additionally, however, we should also prove the statement that the instruction classes abstract all instructions of the architecture (step (4) in figure 5).

Using this 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.

Besides verification purposes, the hierarchical RISC model can also be used for broader applications in a VLSI design environment. For example, the hierarchical structure of the RISC model could be directly used for functional simulation where every specification level builds a virtual machine whose behavior can be verified through simulation techniques. For instance, in [8] a similar structuring of the functional simulation of a commercial RISC processor — the CRISP processor [9] was undertaken using three hierarchical levels (architecture level, pipeline stage level and hardware block implementation). Furthermore, the RISC model can also be useful in a synthesis process where the introduction of a new level by refining existing ones will be guided through the well defined abstractions.

4. Overview of the Verification Method

4.1. Introduction

The general aim of a processor formal verification is to show that every sequence of instructions of any program 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 6), 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. For proving the correctness of a RISC processor, we therefore have to verify that the (pipelined) hardware implies the semantics of every instruction of the instruction set architecture, with respect to possible interactions due to the pipelined execution.

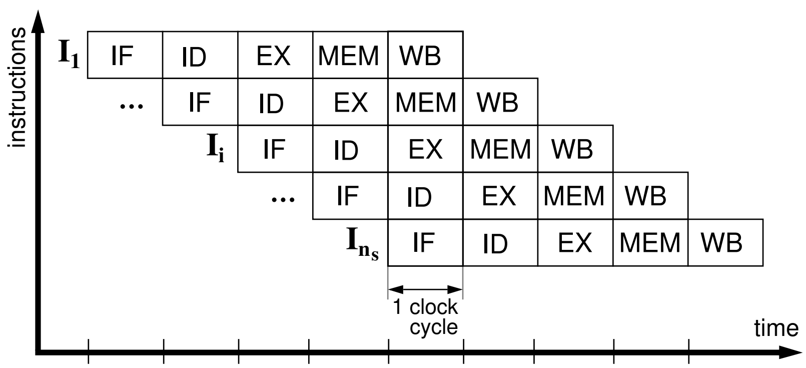

Figure 6. Pipelined Instruction Execution

Referring to the RISC model level described above, our goal is to show that the formal description of the hardware *EBM* implies the formal specification of the architecture level, i.e.

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

In the above correctness statements, the formal specification of the architecture level should include both a specification of data semantics of the instruction set, and a specification of the interactions between any instructions in simultaneous execution in the pipeline. Contemplating on the lifetime of any instruction in the pipeline, we should consider all possible combinations of instruction interactions within the pipeline. The semantics of every single instruction is realized by the sequential execution of its stage operations. On the other hand, due to the simultaneous use of shared resources and the existence of data and control dependencies, the stage and phase instructions within the pipeline could interfere with each other, so that semantical inconsistencies could also occur (see figure 7). In the RISC literature, the inconsistencies that occur in a pipelined execution due to the data and control dependencies and the resources contentions, are called *pipeline conflicts* or *hazards*.

The correctness statement stated above has now to be refined to take the above discussed facts into consideration. In following, we will introduce several formal definitions based on which the correctness statement will be established in the subsequent section.

4.2. *Formal Definitions*

Let *spec* be a predicate representing the specification of the data semantics of a single instruction and let *no_conflicts* be another predicate that is true when no conflicts occur during the simultaneous execution of the instructions specified in its parameters list. We define I_i^p an instruction *i* of a program sequence executed on the pipelined hardware EBM as:

$$I_i^p = spec\ (I_i) \wedge no\_conflicts\ (I_{i-n_s-1},\ldots,\ I_i\ ,\ldots,\ I_{i+n_s-1})$$

where in the concrete example of figure 7 with the five-stage pipeline ($n_s = 5$), we have I_i interfering with $I_{i-n_s-1} = I_{i-4}$ to $I_{i+n_s-1} = I_{i+4}$.

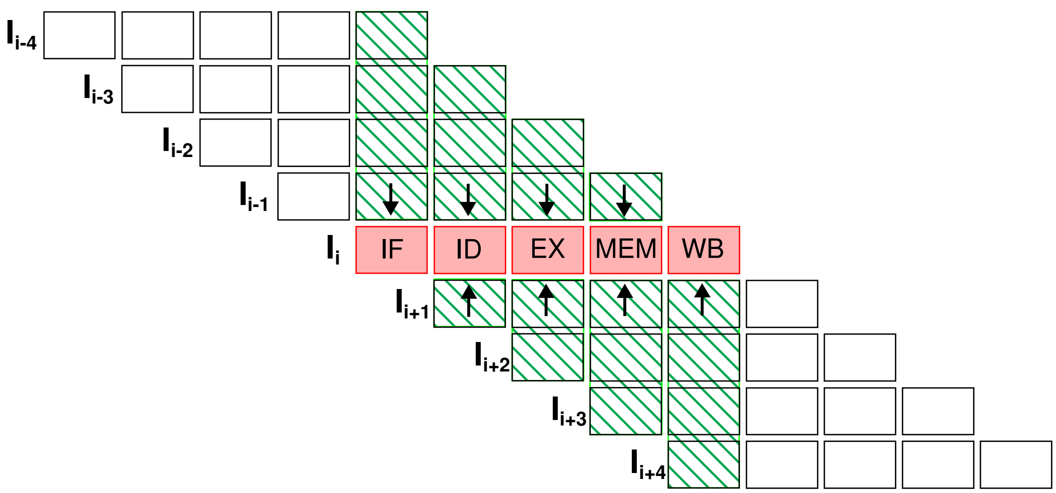

Figure 7. Interactions between Pipelined Instructions

Considering the stage operations of each of the instructions I_{i-n_s-1} to I_{i+n_s-1} that might interfere with I_i , we further define the pipelined stage operations of I_i^p as:

$$S_{i,j}^p = spec\ (S_{i,j}) \wedge no\_conflicts\ (S_{i-n_s-1,n_s},\ \ldots,\ S_{i,j},\ \ldots,\ S_{i+n_s-1,1})$$

where the variable *j* in $S_{i,j}^p$ reflects the order of the pipeline stages, i.e. $S_{i,1}^p = S_{i,IF}^p$, $S_{i,2}^p = S_{i,ID}^p$, etc. In the example of figure 7 the above definition is equivalent to:

$$S_{i,j}^p = spec\ (S_{i,j}) \wedge no\_conflicts\ (S_{i-4,WB},\ \ldots,\ S_{i,j},\ \ldots,\ S_{i+4,IF})$$

Interferences at the phase level are those between parallel stage operations, i.e. for each *j*: $1\ldots n_s$, we consider the combinations of phase operations along the vertical axis of one clock cycle (see dashed box in figure 8. The pipelined phase operations of $S_{i,j}^p$ are hence defined as:

$$\phi_{i,j,k}^p = spec\ (\phi_{i,j,k}) \wedge no\_conflicts\ (\phi_{i-n,j+n,n_p},\ \ldots,\ \phi_{i,j,k},\ \ldots,\ \phi_{i+m,j-m,1})$$

where ($1 \leq n \leq n_s - j$) and ($1 \leq m \leq j$-1). For the example of figure 8 with two-phased clock (n_p =2), a five-stage pipeline ($n_s = 5$) and $j = 4$ (*MEM*), we have ($1 \leq n \leq 1$) and ($1 \leq m \leq 4$-1), i.e. n =1 and

m = 1, 2, 3, meaning that the parallel phases in question range within $\bar{S}_{i-1,4+1}$, $\bar{S}_{i,4}$, $\bar{S}_{i+1,4-1}$, $\bar{S}_{i+2,4-2}$ and $\bar{S}_{i+3,4-3}$, (i.e. $\bar{S}_{i-1,WB}$, $\bar{S}_{i,MEM}$, $\bar{S}_{i+1,EX}$, $\bar{S}_{i+2,ID}$ and $\bar{S}_{i+3,IF}$).

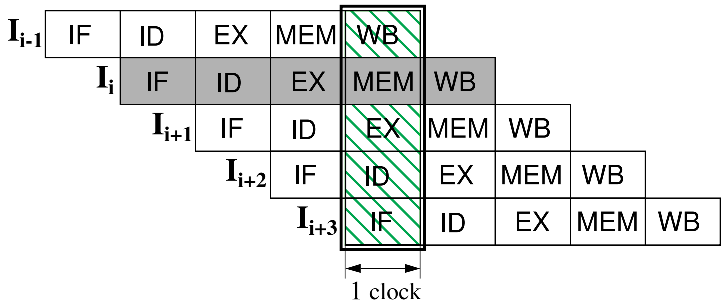

Figure 8. Phase Pipelined Interactions

4.3. *Elaboration of the Correctness Statement*

Starting from the instruction set architecture of a microprocessor, the main correctness proof is to show that the hardware correctly implements the expected semantics of every instruction of the instruction set (i.e. *EBM* $\Rightarrow$ *Architecture Level*). Using the above definitions, this theorem is equivalent to:

$$\vdash \quad \forall\ \bar{I}_i^p.\ EBM \Rightarrow \bar{I}_i^p$$

Based on the RISC model hierarchies introduced in section 3, this correctness statement could be split hierarchically into 3 steps as follows:

Step1: The cumulative effect of the pipelined stage operations implements the semantics of every pipelined instruction correctly (i.e. *Stage Level* $\Rightarrow$ *Architecture Level*):

$$\vdash \quad \forall\ \bar{I}_i^p.\ \bigwedge_{j:1..n_s} \bar{S}_{i,j}^p \Rightarrow \bar{I}_i^p \tag{1}$$

Step2: Every pipelined stage operation is correctly implemented by the cumulative effect of the pipelined clock phase operations (i.e. *Phase Level* $\Rightarrow$ *Stage Level*):

$$\vdash \quad \forall\ \bar{S}_{i,j}^p.\ \bigwedge_{k:1..n_p} \bar{\phi}_{i,j,k}^p \Rightarrow \bar{S}_{i,j}^p \tag{2}$$

Step3: Every pipelined clock phase operation is correctly implemented by the hardware EBM (i.e. *EBM* $\Rightarrow$ *Phase Level*):

$$\vdash \quad \forall\ \bar{\phi}_{i,j,k}^p.\ EBM \Rightarrow \bar{\phi}_{i,j,k}^p \tag{3}$$

Using the definitions of $\bar{I}_i^p$, $\bar{S}_{i,j}^p$ and $\bar{\phi}_{i,j,k}^p$, the above correctness theorems (1), (2) and (3) are equivalent to the following theorems (1'), (2') and (3'), respectively:

$$\vdash \quad \forall\ \bar{I}_i .\ \bigwedge_{j:1..n_s} spec\ (\bar{S}_{i,j}) \wedge no\_conflicts\ (\bar{S}_{i-n_s-1,n_s},\ldots, \bar{S}_{i,j},\ldots, \bar{S}_{i+n_s-1,1}) \tag{1'}$$
$$\Rightarrow spec\ (\bar{I}_i\ ) \wedge no\_conflicts\ (\bar{I}_{i-n_s-1},\ \ldots,\ \bar{I}_i\ ,\ \ldots,\ \bar{I}_{i+n_s-1})$$

$$\vdash \quad \forall\ \bar{S}_{i,j} .\ \bigwedge_{k:1..n_p} spec\ (\bar{\phi}_{i,j,k}) \wedge no\_conflicts\ (\bar{\phi}_{i-n,j+n,n_p},\ldots, \bar{\phi}_{i,j,k},\ldots, \bar{\phi}_{i+m,j-m,1}) \tag{2'}$$
$$\Rightarrow spec\ (\bar{S}_{i,j}) \wedge no\_conflicts\ (\bar{S}_{i-n_s-1,n_s},\ \ldots,\ \bar{S}_{i,j},\ \ldots,\ \bar{S}_{i+n_s-1,1})$$

$$\vdash \quad \forall\ \bar{\phi}_{i,j,k}.\ EBM \Rightarrow spec\ (\bar{\phi}_{i,j,k}) \wedge no\_conflicts\ (\bar{\phi}_{i-n,j+n,n_p},\ldots,\ \bar{\phi}_{i,j,k},\ldots,\ \bar{\phi}_{i+m,j-m,1}) \qquad (3')$$

Each of the theorems (1’), (2’), (3’) can be split into two goals, yielding the following six correctness theorems:

$$\vdash \quad \forall\ \bar{I}_i\ .\ \bigwedge_{j:1..n_s} spec\ (\bar{S}_{i,j}) \Rightarrow spec\ (\bar{I}_i) \qquad \text{(in short } Stage\_Specs \Rightarrow Class\_Specs\text{)} \qquad (1a)$$

$$\vdash \quad \forall\ \bar{I}_i\ .\ \bigwedge_{j:1..n_s} spec\ (\bar{S}_{i,j}) \wedge no\_conflicts\ (\bar{S}_{i-n_s-1,n_s},\ \ldots,\ \bar{S}_{i,j},\ \ldots,\ \bar{S}_{i+n_s-1,1}) \qquad (1b)$$
$$\Rightarrow no\_conflicts\ (\bar{I}_{i-n_s-1},\ \ldots,\ \bar{I}_i,\ \ldots,\ \bar{I}_{i+n_s-1})$$

$$\vdash \quad \forall\ \bar{S}_{i,j}.\ \bigwedge_{k:1..n_p} spec\ (\bar{\phi}_{i,j,k}) \Rightarrow spec\ (\bar{S}_{i,j}) \qquad \text{(in short } Phase\_Specs \Rightarrow Stage\_Specs\text{)} \qquad (2a)$$

$$\vdash \quad \forall\ \bar{S}_{i,j}.\ \bigwedge_{k:1..n_p} spec\ (\bar{\phi}_{i,j,k}) \wedge no\_conflicts\ (\bar{\phi}_{i-n,j+n,n_p},\ldots,\ \bar{\phi}_{i,j,k},\ldots,\ \bar{\phi}_{i+m,j-m,1}) \qquad (2b)$$
$$\Rightarrow no\_conflicts\ (\bar{S}_{i-n_s-1,n_s},\ \ldots,\ \bar{S}_{i,j},\ \ldots,\ \bar{S}_{i+n_s-1,1})$$

$$\vdash \quad \forall\ \bar{\phi}_{i,j,k}.\ EBM \Rightarrow spec\ (\bar{\phi}_{i,j,k}) \qquad \text{(in short } EBM \Rightarrow Phase\_Specs\text{)} \qquad (3a)$$

$$\vdash \quad \forall\ \bar{\phi}_{i,j,k}.\ EBM \Rightarrow no\_conflicts\ (\bar{\phi}_{i-n,j+n,n_p},\ldots,\ \bar{\phi}_{i,j,k},\ldots,\ \bar{\phi}_{i+m,j-m,1}) \qquad (3b)$$

where we used the simple rule “$[(A_1 \Rightarrow \mathrm{A}_2) \wedge (A_1 \wedge B_1 \Rightarrow B_2)] \Rightarrow [(\mathrm{A}_1 \wedge B_1) \Rightarrow (\mathrm{A}_2 \wedge B_2)]$” to derive the theorems (1a), (1b), (2a) and (2b).

According to the nature of these six theorems, in following, we call the three correctness statements (1a), (2a) and (3a) as the *pipeline data* correctness, and the three statements (1b), (2b) and (3b) as the *pipeline control* correctness. By the pipeline data verification, we are concerned with the proof that the cumulative effect of pipeline suboperations yields the data semantics of the given instruction, i.e. verification of the correctness of the pipeline data operations (datapath). While by the pipeline control verification, we are concerned with interferences between the different instructions and suboperations in the pipeline, i.e. the verification of the pipeline (conflict) control of the executed instructions (control unit). These two major correctness steps are briefly discussed in the following subsections and elaborated later in sections 7 and 8, respectively.

4.4. Pipeline Data Verification

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 6) and prove the following — *EBM* $\Rightarrow$ *Phase_Specs* $\Rightarrow$ *Stage_Specs* $\Rightarrow$ *Class_Specs*. Later, these proofs are instantiated for each instruction at the architecture level yielding *EBM* $\stackrel{1}{\Rightarrow}$ (*instantiated*) *Phase Level* $\stackrel{2}{\Rightarrow}$ (*instantiated*) *Stage Level* $\stackrel{3}{\Rightarrow}$ (*instantiated*) *Class Level* = *Architecture Level*. The verification tasks of the pipeline data 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. The verification of the pipeline data process is described in section 6.

4.5. Pipeline Control Verification

The pipeline control correctness consists in the proof that all possible combinations of n_s instructions, within the pipeline, are executed without interference, i.e. no conflicts occur between he different instructions and suboperations in the pipeline. In the open literature on computer architecture, pipeline conflicts are classified into three categories, namely, *resource*, *data* and *control conflicts* (*hazards*) [47]. Accordingly, each of the predicates *no_conflict* at the different levels of the RISC model is hence defined as the following conjunction, where we assign a specific predicate to each kind of conflict, i.e. *Resource_Conflict*, *Data_Conflict* and *Control_Conflict*. Formally:

$$\vdash_{def} \; \mathit{no\_conflict}\,(\ldots) := \; \begin{aligned}[t] (\neg\; & \mathit{Resource\_Conflict}\,(\ldots) \wedge \\ \neg\; & \mathit{Data\_Conflict}\,(\ldots) \wedge \\ \neg\; & \mathit{Control\_Conflict}\,(\ldots)) \end{aligned}$$

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

It is here important to note that this definition is necessary but not sufficient to obtain a total pipeline control correctness since we cannot guarantee that these three classes of conflict exclusively cover all possible conflicts within the pipeline. However, even though no total correctness for all possible pipeline interactions is ensured, we believe that they cover the major properties of known pipeline interactions.

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 will be reduced by an order of magnitude when the notion of classes (as described in section 3.2) 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. The description of the pipeline conflict verification is handled in section 8.

4.6. Verification of Specific Hardware Properties

A RISC processor generally includes some hardware behaviors whose specifications and implementations are processor specific, such as hardware interrupts, cache control, virtual memory (TLB) control, etc. which are part of the RISC processor core [47]. Therefore, in addition to the specifications of the described model levels (cf. section 6), one has to specify the intended behavior formally in form of a predicate, whose correctness has to be implied from the implementation EBM. For example, let *INTR_SPEC* be a predicate that describes the behavior 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 behavior is:

$$\vdash \; \mathit{EBM} \Rightarrow \mathit{INTR\_SPEC}$$

In contrast to the pipeline data and control verifications, the verification of these specific behaviors are non-hierarchical and, in addition, are highly implementation dependent making the development of generalized proof templates within our hierarchical methodology impossible. For instance, different RISC processors implement interrupts in different ways, i.e., whether they are precise or imprecise, maskable or unmaskable, resuming or terminating, etc. [47]. The hardware specific verifications are therefore handled separately in addition to the hierarchical steps of the pipeline data and control verifications and will be discussed in section 9.

5. Temporal Abstraction

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, behavioral abstraction, data abstraction and temporal abstraction [59]. Before getting into the details of the formal specifications and verification using the model levels, we will, in this section, first briefly discuss the structural, behavioral and data abstractions and then we will focus on a new concept of temporal abstraction.

Structural and behavioral 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 behavioral 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 behavioral 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 [53, 83] uninterpreted data types were used. Here we use concrete data types of bit-vectors, e.g. from the bit-vector library of HOL [87]. 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. In contrast to structural and behavioral abstractions, temporal abstraction in pipelined processors is more difficult to tackle as we will see in short.

Temporal abstraction relates the different time granularities which occur in the formal specifications at various levels of abstractions [59]. 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 9).

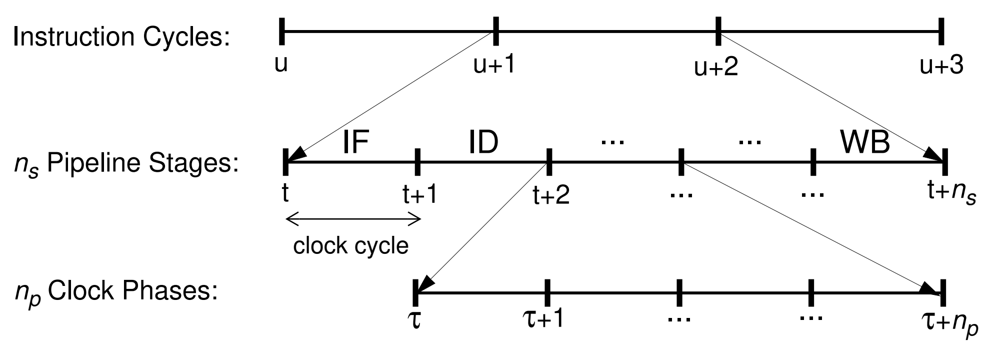

Figure 9. Time Granularities of the RISC Model

Temporal abstraction allows us to hide the unnecessary details at higher levels of abstractions [59]. 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 10). On the other hand, in a pipelined machine (as is the case for RISC processors), instructions are executed in an overlapped manner (figure 10), 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.

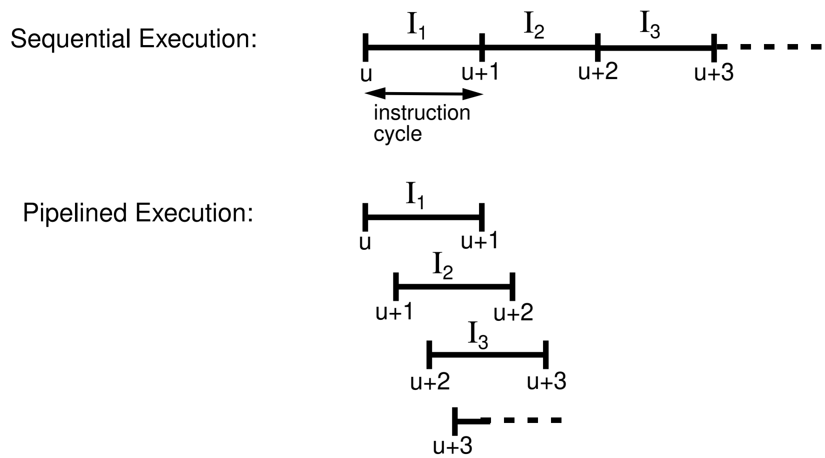

Figure 10. Sequential and Pipelined Executions

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[3]:

$$ADD\_SPEC := \forall \mathbf{u}: Inst\_cycle.\ RF(\mathbf{u+1})[rd(\mathbf{u})] = RF(\mathbf{u})[rs1(\mathbf{u})] + RF(\mathbf{u})[rs2(\mathbf{u})] \quad \text{(i)}$$

Referring to table 3, the ADD-instruction, using the more concrete time granularity of clock cycles (represented by the variable "*t*"), can be specified formally as:

$$ADD\_IMP := \forall \mathbf{t}: Clock\_cycle.\ RF\ (\mathbf{t+5})[rd(\mathbf{t})] = RF(\mathbf{t+1})[rs1(\mathbf{t})] + RF(\mathbf{t+1})[rs2(\mathbf{t})] \quad \text{(ii)}$$

A mapping function that relates abstract time scales in (i) to concrete ones in (ii) is not linear since a unit of time on the abstract time scale does not necessarily correspond to one discrete time

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

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 hardware the addresses *rs1*, *rs2* and *rd* are carried out along the pipeline via a number of instruction registers (*IR*, *IR1*, *IR2* and *IR3*) that keep history of the instruction word fields requested for later use in the pipeline (see figure 11).

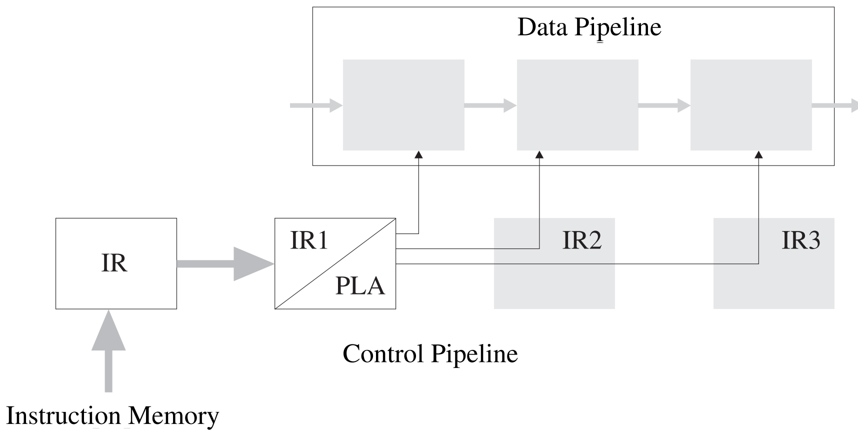

Figure 11. Control Information Flow Through the Instruction Registers

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 [75]. 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 behaviors 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 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* is defined formally as follows[4]:

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

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 and stage levels. 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 7). For example, with $n = n_s = 5$, $C = [read, ID]$, $ORD(read) = 0$ and $ORD(ID) = 1$, we obtain $Time\_abs\ (n_s, [read, ID],\ u) = \frac{1}{5} * (u - 0) + 1 + 0 = \frac{1}{5} u + 1 = t{+}1$. More details about the derivation and implementation of this temporal abtraction function is reported in [76]. The use of the temporal abstraction function *Time_abs* will be illustrated in section 7 while describing the verification process.

6. 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 using 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.

6.1. 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:

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

$$BRZ := \; if\ (RF\ [rs1] = 0) \;\; then \;\; PC \leftarrow PC + offset16 \\ \qquad\qquad\qquad\qquad\quad else \;\; PC \leftarrow PC + 4$$

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

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

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

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

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

6.2. 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 3) 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:

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

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 a unit of time for an instruction cycle, the semantics for the ALU and CONTROL class instructions can be specified formally by the following predicates:

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

5. The expression "$a \longrightarrow\ b\ |\ c$" is an abbreviation for "if a then b else c".

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

6.3. 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 3 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 3) and encoded formally. For example, the ID-stage instruction ID_A of the ALU class (see row ID and column ALU in table 3) is specified by the following predicate:

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

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 semantics specification of the control instructions, the ID-stage instruction ID_C of the CONTROL-class (see row ID and column CONTROL in table 3) is specified as follows:

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

6.4. 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-off from the pipeline architecture (table 3) and encoded formally. For phase transitions that are not explicitly marked in the pipeline architecture, e.g. between the registers *IR* and *IR1* 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:

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

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

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:

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

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

6.5. Electronic Block Model

While the abstract levels of our interpreter model are behavioral 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 hierarchically structured. At the top most level, the EBM is composed 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 12 which illustrates the file structure of the EBM hardware description of DLX in HOL. The underlying layers are a library for macro cells (DLX components) *DLXlib* and the CADENCE standard cell library *GEN_ES2*.

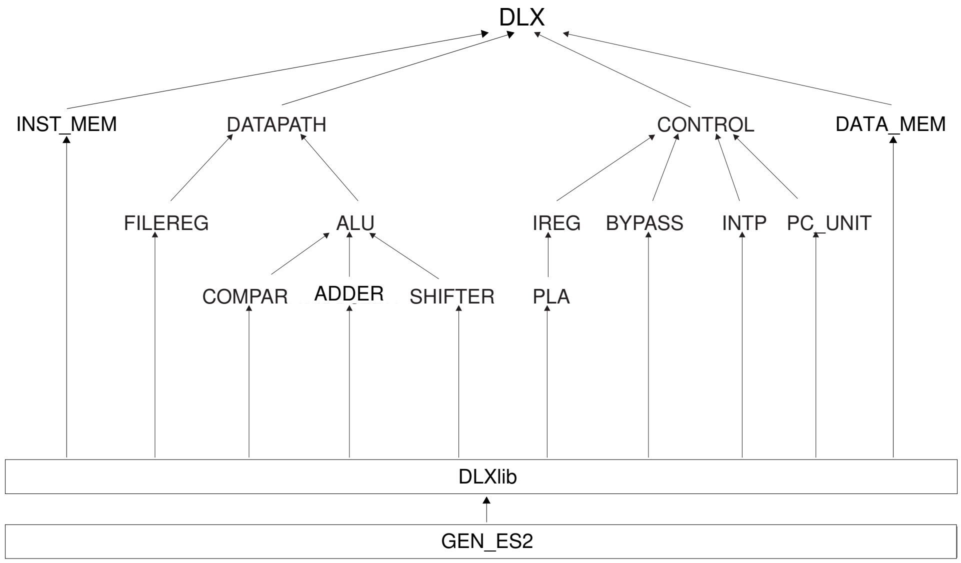

Figure 12. Hierarchical Files Structure of the EBM Hardware Description

Formally, the EBM is specified à la Hanna and Daeche [45] 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 modeled using existential quantification. The top level implementation of the EBM given in figure 2 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_int, ackn, clk1, clk2*) :=

∃ *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*) ∧

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*) ∧

Instr_Memory (*I-MEM, imem_addr, imem_data, clk2*) ∧

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, 53, 83]. In contrast, within our approach the implementation description is completely specified down to the gate level. Since our methodology is embedded within the MEPHISTO [56] verification framework, the formal description of the circuit can either be obtained automatically from an EDIF [35] output of a schematic representation within a CAD tool or from a VHDL description [56]. 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 [55].

6.6. *Use of the Model Formalization*

In general, a formal description of a hardware system could be used as a specification for verification, simulation or synthesis [37]. 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 [21] 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 [46]. This approach of formal synthesis incorporates formal verification with the design process.

7. Pipeline Data Verification

In the pipeline data verification 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 3.2) i.e.:

$$EBM \Rightarrow Phase\_Specs \Rightarrow Stage\_Specs \Rightarrow Class\_Specs$$

and then through instantiation, we show the correctness of the architecture level. Each of these implications can be proven independently. Hence, the pipeline data verification is broken according to the abstraction levels into the following steps:

$$Stage\_Specs \Rightarrow Class\_Specs$$
$$Phase\_Specs \Rightarrow Stage\_Specs$$
$$EBM \Rightarrow Phase\_Specs$$

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 [83] 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 then show how the instantiations are handled. The verification techniques described will be illustrated by some simple examples based on the DLX processor.

7.1. Stage to Class Level

In order to show the correctness of the class level, we prove that individual class specifications are correctly implemented by the sequential execution of their corresponding stage instructions:

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

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 5). 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 5) 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 6.2 and 6.3, respectively, the verification goal for the ALU-class example looks formally as follows:

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

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:

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

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 predicate of the class instruction, the corresponding stage specification predicates, and the list of needed context parameters (including read/write information and an indication for 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 $\mathcal{T}$:

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

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 3), 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*].

7.2. Phase to Stage Level

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

$$\phi_1\_Spec \wedge \ldots \wedge \phi_{n_p}\_Spec \Rightarrow Stage\_Spec$$

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 6.3 and 6.4, respectively, the verification goal for the ID-stage of the ALU-class is given below:

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

Such verification goals for the stage level can also be set automatically using the existing 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:

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

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:

$\mathcal{T}$(n_p,
ID_A_SPEC,
[$\phi_1 ID_A\_SPEC$, $\phi_2 ID_A\_SPEC$],
[[*write*, ϕ_2], [*write*, ϕ_2], [*read*, ϕ_2], [*read*, ϕ_1], [*write*, ϕ_2]])

The context variables required for the proof are easily derived from the implemented pipeline structure (table 3), 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*, ϕ_2].

7.3. EBM to Phase Level

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 behavioral one. However, due to the advantage of having used the hierarchical RISC 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 6.4. 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 [56]. 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, phase instructions including class abstraction functions obviously 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 a 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* = T and *clk2* = F. Let $\mathcal{G}$ be this goal generation function, using the specifications of the phase level and EBM, as described in sections 6.4 and 6.5, respectively, the following parametrization example of $\mathcal{G}$:

$\mathcal{G}$ ($\phi_2 ID_A\_SPEC$, ϕ_2, *EBM*, [])

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

$$\vdash \; EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{F}, \mathbf{T}) \Rightarrow \phi_2 ID_A\_SPEC\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA)$$

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 $\mathcal{g}$ is parameterized as follows:

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

to generate the following three goals:

$$\vdash EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{T}, \mathbf{F}) \Rightarrow \phi_1 ID_C\_SPEC\;\; f_{JMP}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA)$$

$$\vdash EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{T}, \mathbf{F}) \Rightarrow \phi_1 ID_C\_SPEC\;\; f_{JR}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA)$$

$$\vdash EBM\,(PC, I\text{-}MEM, \ldots, A, B, \ldots, BTA, \ldots, ackn, \mathbf{T}, \mathbf{F}) \Rightarrow \phi_1 ID_C\_SPEC\;\; f_{BRZ}\,(A, B, \ldots, PC, RF, \ldots, IR, \ldots, BTA)$$

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 is done automatically using an appropriate general proof scheme 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.

7.4. 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 architecture 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 5). For the ADD-Instruction we obtain the following theorem:

$$\vdash \; ADD\_SPEC\,(PC, RF, I\text{-}MEM, D\text{-}MEM) = ALU\_SPEC\;\; \boldsymbol{add}\,(PC, I\text{-}MEM, RF, D\text{-}MEM)$$

Furthermore, since the correctness proofs at the class level involve a 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 7.1) 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:

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

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 7.3), we use the proven theorems for the phase instructions, e.g.:

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

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:

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

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

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

7.5. 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:

$$\vdash EBM \Rightarrow Architecture\_Specs$$

8. Pipeline Control Verification

In this section, we focus our attention on the pipeline control verification. As discussed in section 4.5, this corresponds to the verification of pipeline conflicts where we emphasize in this paper on the resource, data and control conflicts, which represent the major properties of the pipeline control. 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 automatically generated and explicitly stated, so that the designer can easily formulate the conflict resolution mechanisms either in hardware or in software.

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 will 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.

8.1. 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 3, 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 mechanically extracted from the formal specifications of the model levels (cf. section 6).

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

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

- 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. $Stage\_Rank\ (ID) = 1$, $Phase\_Rank\ (\phi_1) = 0$. 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*)

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

The Predicates *Stage_Used* and *Phase_Used* are automatically extracted from the specifications of the stage and phase level instructions at the clock cycle and clock phase time granularities, respectively (refer to section 6). Each of these predicates is generated as a theorem for the given combination of stage or phase instruction, pipeline stage or clock phase and resources corresponding to a particular level, e.g.:

$$ID_C\_SPEC \vdash Stage\_Used\ (ID_C, PC) = True$$

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*) [54] by a given class or stage instruction at a given pipeline stage or clock phase, respectively:

  Stage_Domain: ((*class_instruction*, *pipeline_stage*, *CL_resource*) $\rightarrow$ *bool*)

  Stage_Range: ((*class_instruction*, *pipeline_stage*, *CL_resource*) $\rightarrow$ *bool*)

  Phase_Domain: ((*stage_instruction*, *clock_phase*, *CL_resource*) $\rightarrow$ *bool*)

  Phase_Range: ((*stage_instruction*, *clock_phase*, *CL_resource*) $\rightarrow$ *bool*)

The Predicates *Stage_Domain/Range* and *Phase_Domain/Range* are also mechanically extracted from the specifications of the stage and phase level instructions at the clock cycle and clock phase time granularities, respectively (refer to section 6). As the case of the above *Stage/Phase_Used* predicates, 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, e.g.:

$$ID_C\_SPEC \vdash Stage\_Domain\ (ALU, ID, RF) = True$$

$$\phi_2 ID_C\_SPEC \vdash \text{Phase\_Range}\ (ID_C, \phi_2, D\text{-}MEM) = False$$

which mean 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 3).

All theorems on the predicates *Stage/Phase_Used*, *Stage/Phase_Domain* and *Stage/Phase_Range* are created once and collected as rules in appropriate lists which will be used later for rewriting 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 6) and generate the required theorems from the formal specifications.

8.2. Resource Conflicts

Resource conflicts (also called *structural hazards* [47, 54, 61, 72] or *collisions* [54, 72]) 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.

8.2.1. Resource Conflict Specification

Referring to the hierarchical RISC model, we will handle the formal specifications of resource conflicts 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. [77] where a compact formalization for post-design verification is presented.

Class Level Conflicts. The resource conflict predicate *Resource_Conflict*, as mentioned in section 4.5, is equivalent to a multiple conflict between the maximal number of class instructions present in the pipeline (figure 6), i.e.:

$$\vdash_{def} \textit{Resource\_Conflict}\ (\ldots) := \textit{Multiple\_Stage\_Conflict}\ (\ldots)$$

This *Multiple_Stage_Conflict* predicate is true if any pair of the corresponding stage instructions compete for one resource (see hatched box in figure 13). Formally, *Multiple_Stage_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.):

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

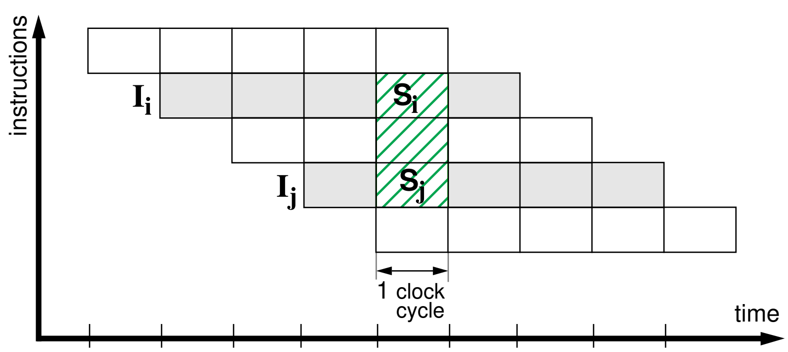

Figure 13. Stage Resource Conflict

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 13), 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:

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

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 14).

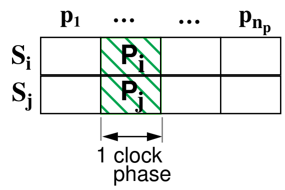

Figure 14. Phase Resource Conflict

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.:

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

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. $\varphi_1 = \phi_1$, $\varphi_2 = \phi_2$):

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

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 14) 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:

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

8.2.2. Resource Conflict Verification

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

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

With the definition of the resource conflicts at the different levels of the hierarchical RISC model, this goal is split into two parts as:

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

and

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

Given the fact that the definition of each level relies on the data semantics specifications including the information about the use of the resources, we use the results of section 7 (i.e. *EBM* ⇒ *Phase_Specs* ⇒ *Stage_Specs*) rewritten these two goals to:

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

and

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

Stage Level Conflicts. Using the definition of *Multiple_Stage_Conflict*, the expansion of the first 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_Stage_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:

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

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

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

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.

In the proof of the first step, we expand the *Stage_Used* predicates and use the theorem implied by the stage specifications (*Stage_Specs*). The proof 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:

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

Phase Level Conflicts. 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:

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

Using the definition of *Multiple_Phase_Conflict*, this goal is expanded in terms of dual phase conflicts. According to the theorems implied by the phase specifications (*Phase_Specs*), one obtains either *True* (which means conflict freedom) or a number of subgoals of the form:

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

In this case, the resource conflict remains and the phase level (implementation EBM) has to be changed appropriately, e.g. by using an additional buffer. 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:

$$Phase\_Specs \vdash \forall\, \phi_i\, \phi_j : phase\_instruction\,.\ \neg\ Dual\_Phase\_Conflict\ (\phi_i, \phi_j)$$

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 phase and stage level implementation which ensure 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.:

$$Phase\_Specs, Stage\_Specs \vdash \forall\, I_i\ \ \neg\, Resource\_Conflict\ (\ldots)$$

Using previous results of section 7 that *EBM* ⇒ *Phase_Specs* ⇒ *Stage_Specs*, we obtain:

$$EBM \vdash \forall\, I_i\ \ \neg\, Resource\_Conflict\ (\ldots)$$

8.3. Data Conflicts

Data conflicts (also called *data hazards* [47, 54, 72], *timing hazards* [61], *data dependencies* [39] or *data races* [36]) 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) [39, 47, 54, 72] (also called destination source (DS), source destination

(SD) and destination destination (DD) conflicts [61]). 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 of multicycles operations or by multiple instructions issue [47]. 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.

8.3.1. 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 will be 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 [77].

Class Level Conflicts. Considering the lifetime of any instruction I_i (see figure 7), the data conflict predicate, i.e. *Data_Conflict*, should involve all the instructions that could lead to data conflicts. The predicate *Data_Conflict* is thus defined in terms of a multiple data conflict predicate, which includes the ($2\,n_s$ - 1) instructions $\bar{I}_{i-n_s-1} \ldots \bar{I}_{i+n_s-1}$ with corresponding sequential fetch times $\bar{t}^0_{i-n_s-1} \ldots \bar{t}^0_{i+n_s-1}$ that could interfere with I_i fetched at time t_i^0, i.e.:

$$\vdash_{def}\ Data\_Conflict := Multiple\_Data\_Conflict\ (\bar{I}_{i-n_s-1},\ldots,\ \bar{I}_{i+n_s-1})$$

The predicate *Multiple_Data_Conflict* is true whenever any two class instructions conflict on some data. Hence, we define *Multiple_Data_Conflict* as the disjunction of all possible dual data conflicts (represented by *Dual_Data_Conflict*) as follows:

$$\begin{array}{l}\vdash_{def}\ Multiple\_Data\_Conflict\ (\bar{I}_{i-n_s-1},\ldots,\ \bar{I}_{i+n_s-1}) :=\\ \quad \exists\ \bar{t}^0_{i-n_s-1} \ldots \bar{t}^0_{i+n_s-1} : Clock\_cycle.\\ \qquad \bigvee\limits_{\substack{i,\,j\\ (i,\,j\,=\,i-n_s-1 \ldots\, i+n_s-1)\\ (i\,<\,j)}} Dual\_Data\_Conflict\ ((I_i,\ t_i^0),\ (I_j,\ t_i^0 + j - 1))\end{array}$$

The predicate *Dual_Data_Conflict* is true, if there exists a resource of the programming model (class level) for which two class instructions I_i and I_j fetched at time points t_i^0 and t_j^0, respectively, conflict. Further, conforming to our RISC model, we handle the *Dual_Data_Conflict* 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:

$\vdash_{def}$ *Dual_Data_Conflict* $((I_i,\ t_i^0), (I_j,\ t_j^0)) :=$
$\exists\, r$ **:** *CL_resource* **.**
Stage_Data_Conflict $((I_i,\ t_i^0), (I_j,\ t_j^0), r)$

Stage Level Conflicts. Let I_i be an instruction that is fetched 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 fetched 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 15). 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 happens, 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 8.1)). Hence, the timing condition for the RAW conflict, i.e. ($t_j^u \leq t_i^u$), is equivalent to $(t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))$.

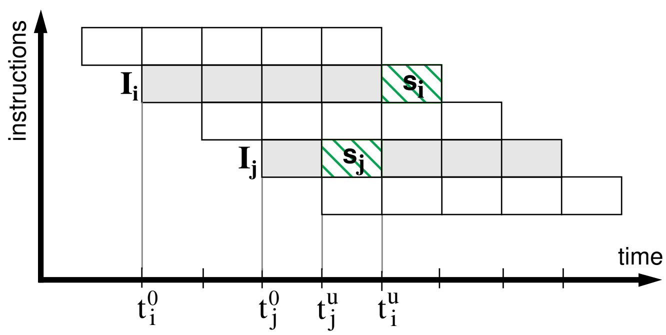

Figure 15. RAW Data Conflict

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

$\vdash_{def}$ *Stage_RAW_Conflict* $((I_i,\ t_i^0), (I_j,\ t_j^0), r):=$
$\exists\, s_i\, s_j$ **:** *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:

$$\vdash_{def} \textit{Stage\_WAR\_Conflict}((I_i,\ t_i^0),(I_j,\ t_j^0),r) :=$$
$$\exists\, s_i\, s_j : \textit{pipeline\_stage}\,.$$
$$(0 < (t_j^0 - t_i^0)) \wedge$$
$$((t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))) \wedge$$
$$\textit{Stage\_Domain}\ (I_i,\ s_i,\ r) \wedge$$
$$\textit{Stage\_Range}\ (I_j,\ s_j,\ r)$$

$$\vdash_{def} \textit{Stage\_WAW\_Conflict}((I_i,\ t_i^0),(I_j,\ t_j^0),r) :=$$
$$\exists\, s_i\, s_j : \textit{pipeline\_stage}\,.$$
$$(0 < (t_j^0 - t_i^0)) \wedge$$
$$((t_j^0 - t_i^0) \leq (\theta(s_i) - \theta(s_j))) \wedge$$
$$\textit{Stage\_Range}\ (I_i,\ s_i,\ r) \wedge$$
$$\textit{Stage\_Range}\ (I_j,\ s_j,\ r)$$

Phase Level Conflicts. A special case of the data conflict timing condition arises when a resource is simultaneously used by the instructions I_i and I_j, i.e. $t_i^u = t_j^u$. In this situation, the data conflict should be examined at the phase level. Let S_i and S_j be 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 16, 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 initiate time $\tau_i^0 = \tau_j^0 = \tau^0$ (figure 16), 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 8.1). 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:

$$\vdash_{def} \textit{Phase\_RAW\_Conflict}\ (S_i,\ S_j,\ r) :=$$
$$\exists\, p_i\, p_j : \textit{clock\_phase}.$$
$$(\xi(p_j) < \xi(p_i)) \wedge$$
$$(\theta(\vartheta(S_j)) < \theta(\vartheta(S_i))) \wedge$$
$$\textit{Phase\_Range}\ (S_i,\ p_i,\ r) \wedge$$
$$\textit{Phase\_Domain}\ (S_j,\ p_j,\ r)$$

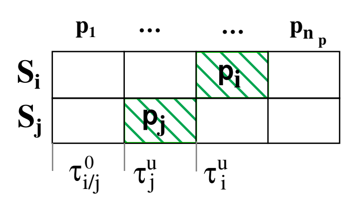

Figure 16. Phase RAW Data Conflict

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

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

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

8.3.2. 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.:

$$
\vdash\ \neg\ \mathit{Data\_Conflict} \quad \Leftrightarrow \quad
\begin{array}{l}
(\neg\ \mathit{RAW\_Conflict}\ \land \\
\ \neg\ \mathit{WAR\_Conflict}\ \land \\
\ \neg\ \mathit{WAW\_Conflict})
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{EBM} \vdash \forall\ I_{i-n_s-1}, \ldots, I_{i+n_s-1} : \mathit{class\_instruction}. \\
\qquad\qquad \neg\ \mathit{Multiple\_RAW\_Conflict}\ (\bar{I}_{i-n_s-1}, \ldots, \bar{I}_{i+n_s-1})
\end{array}
$$

This goal includes a quantification over all possible conflict combinations that could occur between all permutations of (2 n_s - 1) 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:

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

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

$$
\begin{array}{l}
\vdash\ (\forall\ I_i\, I_j : \mathit{class\_instruction}. \\
\qquad \forall\ t_i^0\ t_j^0 : \mathit{Clock\_cycle}. \\
\qquad\quad \neg\, \mathit{Dual\_RAW\_Conflict}\ ((I_i,\ t_i^0), (I_j,\ t_j^0))) \\
\quad \Rightarrow (\forall\ \bar{I}_{i-n_s-1}, \ldots, \bar{I}_{i+n_s-1} : \mathit{class\_instruction}. \\
\qquad\quad \neg\ \mathit{Multiple\_RAW\_Conflict}\ (\bar{I}_{i-n_s-1}, \ldots, \bar{I}_{i+n_s-1}))
\end{array}
$$

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.

Stage Level Conflicts. Using the definition of *Dual_RAW_Conflict* (cf. section 8.3.1), the goal for the above step 1 is equivalent to:

$$EBM \vdash \forall I_i\, I_j : class\_instruction\,.\ \forall t_i^0\ t_j^0 : Clock\_cycle\,.\ \forall r : CL\_resource\,.\ \neg\ Stage\_RAW\_Conflict\,((I_i,\ t_i^0), (I_j,\ t_j^0), r)$$

Since the definition of stage level conflict relies on the data semantics specifications of the stage level, we would obtain the same result by proving the goal:

$$Stage\_Specs \vdash \forall I_i\, I_j : class\_instruction\,.\ \forall t_i^0\ t_j^0 : Clock\_cycle\,.\ \forall r : CL\_resource\,.\ \neg\ Stage\_RAW\_Conflict\,((I_i,\ t_i^0), (I_j,\ t_j^0), r)$$

and rewrite it using previous results of section 7 that *EBM* ⇒ *Phase_Specs* ⇒ *Stage_Specs*.

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:

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

where the number "3" corresponds to the difference $\theta(s_i) - \theta(s_j)$ = "$\theta(WB) - \theta(ID)$".

This result is interpreted as follows: as long as the fetch times of the conflicting LOAD and ALU-instructions satisfy the condition "$(t_j^0 - t_i^0) \le 3$", there exists a data conflict. In order to resolve it, we should be able to neutralize this timing condition by either modifying the stage level (hardware EBM) or using some hardware or software conflict resolution techniques. Here the timing information "$(t_j^0 - t_i^0) \le 3$" gives an exact reference for the number of pipeline slots that have to be tackled, namely a three clock cycles penalty.

Phase Level Conflicts. A special case of the above condition is where "$(t_j^0 - t_i^0) = 3$". With $t_i^u = (t_i^0 + \theta(WB))$ and $t_j^u = (t_j^0 + \theta(ID))$ (cf. section 8.3.1), 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 8.3.1, the data conflict should be explored at the lower time granularity of the phase level by setting the following goal:

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

and checking the conflict freedom at the phase level. If the goal is proven correct, no data conflict happens for $(t_j^0 - t_i^0) = 3$, otherwise the conflict pertains. Furthermore, since the phase level involves all resources of the machine, this result could also be reached by a systematic check of all RAW data conflicts at the phase level. This is then done by setting the following goal:

$$Phase\_Specs \vdash \forall S_i\, S_j : stage\_instruction\, .\ \forall r : CL\_resource\, .\ \neg\, Phase\_RAW\_Conflict\,(S_i, S_j, r)$$

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

For data conflict resolution, there exists a variety of techniques. We list here three of them, namely, software scheduling, data bypassing and pipeline interlocking (stalling). While the first is done in software, i.e. the conflict is resolved by the programmer (compiler), the latter two are implemented in hardware and are hence transparent to the programmer. Recent pipelined processors with multicycles functional units and multiple instructions issue use more advanced techniques for resolving data hazards such as dynamic scheduling—a hardware solution based on techniques like scoreboarding or reservation stations to support instruction level parallelism [47]. These techniques will not be discussed here and are subject of future verification research.

8.3.2.1. Software Scheduling

Software scheduling (also called *instruction scheduling*) [47] is a simple technique based on the reordering of the instructions of the sequential program, such that no data dependencies occur. Dependent instructions are taking apart by filling pipeline slots with useful instructions from the original code. If no reordering is possible by filling useful instructions, the compiler inserts NOP (no-operation) instructions to fill the pipeline slots. Referring to the above data conflict example, the software scheduling ensures that the fetch time of a LOAD-instruction followed by an ALU-instruction is at least three time units apart. The execution of the ALU instruction is hence delayed till the loaded value is available. Therefore the name *delayed execution* or *delayed load* that is widely used in the literature. Formally, we express the behavior of the software scheduling by means of software *constraints* that lead to the proof of the data conflict goal. For the above example, the required software constraint is defined as:

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

In general, the software constraints represent those conditions which are to be met for reordering (scheduling) the software code, so as to avoid conflicts. Given a list of all software constraints in form of instruction scheduling timing conditions, we are able to prove that for all instruction combinations, no (stage or phase) RAW data conflict could occur, i.e. formally:

$$SW\_Constraints \vdash \neg\, RAW\_Conflict$$

8.3.2.2. Bypassing

The most widely used data conflict resolution technique is *bypassing* (also called *forwarding*) [47]. This simple hardware solution 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 behavior is implemented in hardware by using some registers and corresponding feedback paths that hold and forward this data, respectively and a logic circuitry that controls a set of operand multiplexers.

Let *Bypass_Cond* be a predicate which formalizes the existence of the required buffers and bypass paths. Using the definition of *Stage_Range* and *Stage_Domain*, we easily express this bypass condition *Bypass_Cond* as follows:

$$\vdash_{def} \textit{Bypass\_Cond}\ (I_i, I_j) := \\ \exists\, rb\,.\ (rb = RF) \wedge \textit{Stage\_Range}\ (I_i, EX, rb) \wedge \textit{Stage\_Domain}\ (I_j, EX, rb)$$

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 8.3.1) 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 yielding:

$$\textit{Bypass\_Cond} \ \vdash\ \neg\, \textit{Stage\_RAW\_Conflict}$$

Let *BYPASS_SPEC* be a predicate that describes the intended behavior of the implemented hardware logic for data conflict resolution. This behavioral specification should satisfy all required bypass conditions *Bypass_Conds* as given in the above example, i.e.:

$$\vdash\ \textit{BYPASS\_SPEC} \Rightarrow \textit{Bypass\_Conds}$$

Assuming that we have implied the bypass specification predicate *BYPASS_SPEC* from the hardware implementation EBM in a separate proof step (cf. section 9.1), i.e.:

$$\vdash\ \textit{EBM} \Rightarrow \textit{BYPASS\_SPEC}$$

we obtain:

$$\vdash\ \textit{EBM} \Rightarrow \textit{Bypass\_Conds}$$

and through transitivity:

$$\textit{EBM} \ \vdash\ \neg\, \textit{Stage\_RAW\_Conflict}$$

8.3.2.3. Pipeline Interlocking

Not all potential data hazards can be handled using bypassing. For example consider the previous case of the LOAD followed by an ALU instructions that requires the loaded value. The load instruction does not have the data until the end of the MEM stage and no forwarding is possible. One solution to this problem is a delayed execution (load) using software scheduling as described in a previous section. Alternatively, a hardware solution called *pipeline interlocking* [47] is here applicable. The interlock mechanism causes the pipeline to stall in the case of data conflicts. A *stall* eliminates the data conflict by allowing instructions that were issued earlier than the stalled instruction to proceed and delay all instructions issued after until the data will be available. It behaves as an issuing of *hardware* NOP instructions. The advantage of pipeline interlocks is that they stall the pipeline as long as the data is not available, which is in contrast to software scheduling where the number of slots should be known a priori from the given architecture. Stalls are hence often used in conjunction with cache misses and for load interlocks. The disadvantage however, is the penalty of wasted pipeline slots, i.e. unlike software scheduling, where useful instructions can be inserted into the pipeline slot.

For the timing condition of the above example without bypassing, the pipeline has to be stalled for three clock cycles (with bypassing only one clock cycle). The hardware logic that implements the stalling mechanism should recognize the data conflict situation according to the following condition:

$$\vdash_{def} \; Stall\_Cond := ((I_i = LOAD) \wedge (I_j = ALU) \wedge (r = RF) \wedge (0 < (t_j^0 - t_i^0))$$

and causes a stall. Let *Stall* (*n*) be a function that specifies the hardware behavior of stalling the pipeline for *n* cycles. For the above data conflict example, the property required from the hardware is:.

$$\vdash \; Stall\_Cond \Rightarrow Stall\ (\mathbf{3})$$

Here the function *Stall* delays the use time of the data t_j^u by 3 clock cycles. Because of the linear relationship $t_j^u = t_j^0 + \theta(s_j) = t_j^0 + 3$, the fetch time will be (virtually) incremented by a proportion of 3 and hence the timing condition $(t_j^0 - t_i^0) \leq 3$ will be $(t_j^0 + 3 - t_i^0) \leq 3$. This latter is now equivalent to $(t_j^0 - t_i^0) \leq 0$ which is in contradiction to the timing condition on the left part of the above data conflict example goal, i.e. *Stall_Cond*, and therefore the conflict will no more happen, obtaining:

$$Stall\_Cond \;\vdash\; \neg\, Stage\_RAW\_Conflict$$

The interlock hardware should detect the conflict situation and provide an appropriate stall for all sets of properties of the above form. When an instruction is stalled, the hardware should stall all instructions issued later than the stalled instructions and let all instructions issued later be continued. This behavior along with the set of properties described above builds the interlock specification (*INTERLOCK_SPEC*). As with the *BYPASS_SPEC*, we should be able to imply the set of all stall condition *Stall_Conds* from the *INTERLOCK_SPEC* behavioral specification. This has to be done manually by extracting the properties from the hardware components implementing the interlock mechanism. Assuming that we have shown in a separate proof step that the *EBM* implies the interlock specification predicate *INTERLOCK_SPEC*, we hence yield:

$$\vdash \; EBM \Rightarrow INTERLOCK\_SPEC \Rightarrow Stall\_Conds$$

and through transitivity:

$$EBM \;\vdash\; \neg\, Stage\_RAW\_Conflict$$

Although no further details on the non-hierarchical verification steps described above will be given in this paper (we have not experimented with stalls in our DLX), we believe that we have provided a scheme on how to tackle stalls (interlocks) within the scope of our methodology. Nonetheless, the data conflict proof automation developed in this work mechanically generates of the stall conditions as well as the required minimum number of stalling cycles, providing a great help for checking and identifying the required properties.

In [82] another method on the verification of a pipelined RISC processor similar to the DLX with stalls is presented. This method considers stall cases as part of the overall specification and proof correctness statement and is hence different from the proof approach presented above.

8.3.2.4. *Summary*

For the stage level, we obtained:

$$\textit{Stage\_Specs, Bypass\_Conds, Stall\_Conds, SW\_Constraints} \vdash \neg\, \textit{Stage\_RAW\_Conflict}$$

and for the phase level:

$$\textit{Phase\_Specs, Stall\_Conds, SW\_Constraints} \vdash \neg\, \textit{Phase\_RAW\_Conflict}$$

Using previous results of section 7 that *EBM* ⇒ *Phase_Specs* ⇒ *Stage_Specs*, and since the *Bypass_Conds* and *Stall_Conds* condition sets are all implied from the *EBM* via the *BYPASS_SPEC* and *INTERLOCK_SPEC* specifications, respectively, we have:

$$\textit{EBM}, \textit{SW\_Constraints} \vdash \bigwedge_i \neg\, \textit{Stage\_RAW\_Conflict}$$

and with $\bigwedge_i \neg\, \textit{Stage\_RAW\_Conflict} \Rightarrow \neg\, \textit{Multiple\_RAW\_Conflict}$, we obtain:

$$\textit{EBM}, \textit{SW\_Constraints} \vdash \neg\, \textit{Multiple\_RAW\_Conflict}$$

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

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

and finally:

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

The timing conditions that has been automatically generated by the exhaustive proof of all possible combinations of instructions are of considerable importance to the verifier as well as to the designer of the processor. Regardless of the conflict resolution technique implemented, they point out at different hierarchical levels the exact conflict cases with all related informations on the used resources, the involved class instructions, pipeline stages and clock phases. This information can be used either to refine the hardware or for generating test vectors for a verifications using simulation techniques.

8.4. *Control Conflicts*

Control conflicts (also called *control hazards* [47, 72], *branch hazards* [47], *sequencing hazards* [61] or *branch dependencies* [39]) arise from the pipelining of branches and other instructions that change the program counter *PC*, i.e. interruption of the linear instruction flow.

In 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.

8.4.1. Control Conflict Specification

Let $A_f(I_i,\ t_i^0)$ be the fetch address of an instruction I_i fetched 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 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* [54] and thus the software control conflict could be defined as follows:

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

8.4.2. 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:

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

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

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

Since the fetch times t_i^0 and t_j^0 satisfy $(0 < (t_j^0 - t_i^0))$, the timing condition for the control conflict $((t_j^0 - t_i^0) \le 1)$ is equivalent to $((t_j^0 - t_i^0) = 1)$. Referring to the discussion on data conflict verification in section 8.3.2, 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):

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

For the DLX example, we obtain:

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

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. There are many methods for dealing with this branch penalty. Some are software (compiler) based while others are implemented in hardware. The simplest scheme for handling control conflicts is to *freeze* or *flush* the pipeline [47], holding or deleting any instructions after the branch until the branch destination is known. This simple hardware solution is similar to pipeline interlocking. Another commonly used technique is *delayed branch* using software scheduling [47]. A higher performance and considerably more complex scheme is to *predict* the branch as *taken* or *not-taken* by allowing the hardware to continue fetching the target or next instructions, respectively, regardless of the outcome of the branch. If the prediction is wrong, however, the hardware turns the fetched instruction into a NOP and restarts the fetching at the correct address increasing hence the branch penalty [47]. This branch prediction can be done statically (at compile-time) or dynamically (at run-time). The former is based on a careful software scheduling where the latter is implemented in hardware, for example via the use of a *branch history file* or a *branch target buffer* [34]. All these techniques try to reach a nearly zero-delay branch with the aim of increasing the instruction level parallelism. However, there are different kinds of mechanisms that are implemented in different ways by different processors [58]. Hence, a general formalism cannot be given within the scope of our methodology. Referring to the discussion in section 4.6, the behavior of the hardware implemented freeze or dynamic branch prediction mechanisms has to be specified formally and proven correct from the hardware implementation (as done for example in [50]). Using this formal specification of the resolution technique, the above timing condition for the avoidance of control conflicts has to be implied. Since none of these hardware resolution techniques is implemented in DLX, we have not experimented with these features and no further verification details of these non-hierarchical proof steps will be elaborated in this paper.

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 software constraint required for software scheduling in this case is defined as follows:

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

Delayed branches are an architecturally visible feature of the pipeline. This is the source of their primary advantage by allowing the use of simple compiler scheduling to reduce the branch penalties. For RISC machines as DLX with single-cycle branch delays, the delayed branch approach is attractive since it yields good performance with minimal hardware cost [47]. 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 [47].

More recently, with deeper pipelines and longer branch delays more powerful hardware schemes for branch prediction are adopted. Recent RISC architectures include both delayed and non-delayed branches, relying on hardware *dynamic branch prediction*, *branch folding*, etc. [34]. While in this paper, we presented a verification approach for delayed branches, the handling of the verification of advanced dynamic branch prediction is subject of future work. The results of the current work, however, are universal in the sense that they give, independent of the considered architecture or conflict resolution technique, all the necessary conditions for control hazards situations which will be a starting point for such future work.

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.:

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

8.5. Summary

Our ultimate goal in proving the pipeline control correctness by showing the non-existence of pipeline conflicts, i.e. resource, data and control conflicts has been derived at different levels of the hierarchical RISC model as:

$$\begin{array}{lll} & \vdash & \bigwedge_i \neg\ Stage\_Conflict \Rightarrow \neg\ Multiple\_Conflict\ (...) = \bigwedge_i no\_conflicts\ (I_i) \\ EBM,\ Stage\_Specs & \vdash & \neg\ Stage\_Conflict = \bigwedge_j no\_conflicts\ (S_j) \\ EBM,\ Phase\_Specs & \vdash & \neg\ Phase\_Conflict = \bigwedge_k no\_conflicts\ (\phi_k) \end{array}$$

Given an adequate implementation EBM avoiding mutual resource use and involving conflict resolution mechanisms in hardware and/or in software, we conclude the pipeline control correctness from the theorems yielded in sections 8.2.2, 8.3.2 and 8.4.2, i.e.:

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

The obtained proof of the conflicts freedom has been achieved at an abstract level by ranging over class instructions. Since neither structural, data nor temporal abstraction exist between the architecture and class levels and consequently they involve the same resources and have the same timing behaviors, the obtained theorems for the pipeline correctness can be without circumscriptions transferred to the architecture level yielding the formal proof for the pipeline control correctness for all combinations of architectural instructions.

In contrast to the pipeline data 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*. Besides, with exception of the specification of the processor specific arguments of the enumeration types (cf. section 8.1), all required information about the specific processor that is to be verified are gained through mechanical extraction from the already existing formal specifications of the model levels (cf. section 6).

Furthermore, the verification method presented is constructive in that it helps the designer, within a post-design verification process, in validating existing software and/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 hardware resolution techniques such as bypassing, interlocks (stalls), dynamic branch prediction, or dynamic scheduling.

9. Non-Hierarchical Verifications

A RISC processor generally includes some hardware behaviors whose specifications and implementations are processor specific, such as hardware interrupts, cache control, implementing a bus protocol, handling multicycles operations, etc. which are part of the processor core design. In contrast to the previous pipeline verifications based on the hierarchical model, these specific behaviors are non-hierarchical and cannot be handled automatically within our methodology. In addition, they are highly implementation dependent, e.g., different RISC processors implement interrupts in different ways. For instance, where the forced jump is to be inserted, the manner in which the interrupted program is restarted, etc. can vary differentiating whether the current interrupt is precise or imprecise, maskable or unmaskable, resuming or terminating, etc. [47]. Therefore, in addition to the specifications of the described model levels (cf. section 6), one has to specify the intended (e.g. interrupt) behavior formally in form of a predicate, whose correctness has to be implied manually from the implementation EBM in separate verification steps besides the pipeline data and control hierarchical verifications.

From last section on pipeline control verification, we have seen that extra manual verification steps are necessary to deal with the verification of pipeline conflicts that are resolved in hardware such as bypassing, interlocks (stalls), dynamic branch prediction, or dynamic scheduling. These behaviors are also non-hierarchical and highly implementation dependent. Therefore, no generalized verification templates for these specific behaviors could be provided within the scope of our hierarchical methodology. They fall in the same category as interrupts and other processor specific properties which verification is non-hierarchical.

In following, we briefly describe the specification and verification of the bypassing of DLX and in more details those of the interrupt behaviors at implemented in the DLX processor. Experimental results on these two verifications are given in section 11. We have not experimented with the non-hierarchical verification of other hardware related properties, however, as the current implementation of DLX does not support them.

9.1. Bypassing

The DLX hardware provides a bypass logic which identifies a conflict situation through inspection of the source and destination addressees of the used registers of each instruction within the pipeline. This behavior is implemented in hardware by using some registers and corresponding feedback paths that hold and forward this data, respectively and a logic circuitry that controls a set of operand multiplexers (see figure 17). Referring to the discussion in section 8.3.2.2, the implemented bypass behavior 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 behavior. This predicate specifies how the processor behaves in case of a data conflict by detecting it and forwarding the right data to the right pipeline stage where it is needed. A simplified form of the high-level specification of the intended behavior of the bypass logic looks as follows[7]:

$$
\begin{aligned}
\vdash_{def}\ & BYPASS\_SPEC := \\
& \forall\, t.\ \forall\, op. \\
& \quad \exists\, in\_a\ in\_b. \\
& \quad\quad (ALUOUT\ (t{+}1) = op\ in\_a\ in\_b) \land \\
& \quad\quad (in\_a = (rs1 = rd\_exe) \Rightarrow (ALUOUT\ (t)) \,|\, (rs1 = rd\_mem) \Rightarrow (ALUOUT1\ (t)) \,|\, A(t)) \land \\
& \quad\quad (in\_a = (rs2 = rd\_exe) \Rightarrow (ALUOUT\ (t)) \,|\, (rs2 = rd\_mem) \Rightarrow (ALUOUT1\ (t)) \,|\, B(t))
\end{aligned}
$$

In order to prove that the hardware *EBM* implements this behavior, we shall prove:

$$\vdash\ DATAPATH \land BYPASS\_UNIT \Rightarrow BYPASS\_SPEC$$

where we extract the required bypassing behavior from the datapath and the bypass unit which is part of the control unit of *EBM*. These predicates are expanded to the logic level and verified using a set of proof scripts that has been implemented for the verification of the bypassing mechanism. We finally yield the theorem

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

Unlike the scripts for the hierarchical verifications of the data and control, these are not parametrized and no general template can be provided since the hardware implementations of the bypass logic varies from one processor to another.

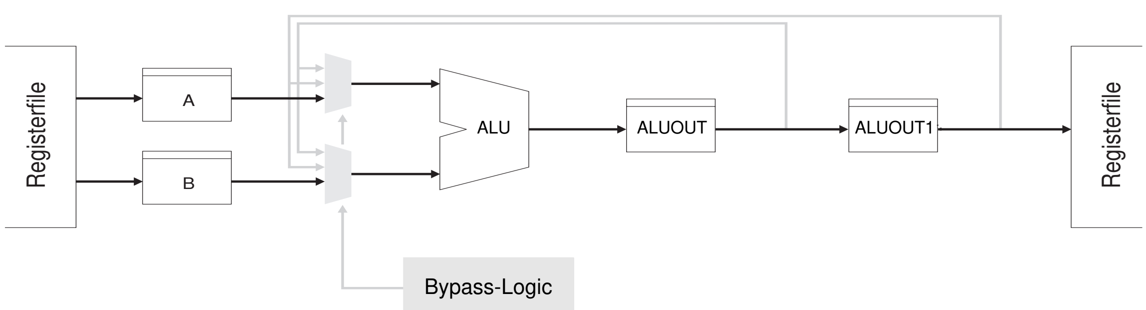

Figure 17. Bypassing Implementation in DLX

9.2. Interrupts

The DLX implements precise interrupt, i.e. all instructions issued into the pipeline before the interrupt occurred will be completed and the pipeline will be restarted after the interrupt handling from the same point where the interrupt occurred. The interrupt handling is implemented via an

7. A detailed formal specification and verification of *BYPASS_SPEC* for a hardware implementation of DLX is reported elsewhere [31].

interrupt pipeline (figure 18), which stores internal exceptions and forwards them till the WB stage. There, all exceptions will be treated at once with the priority of their precedence. External interrupts have highest priority. When an interrupt is recognized, the processor jumps to a specific vector address of an interrupt service routine. Since here the program counter PC is manipulated, control conflicts can occur when a control instruction or a successor instructions in the respective delay slot is interrupted. These conflicts are solved in hardware by detecting these situations and ensuring that the pipeline be restarted from a safe point.

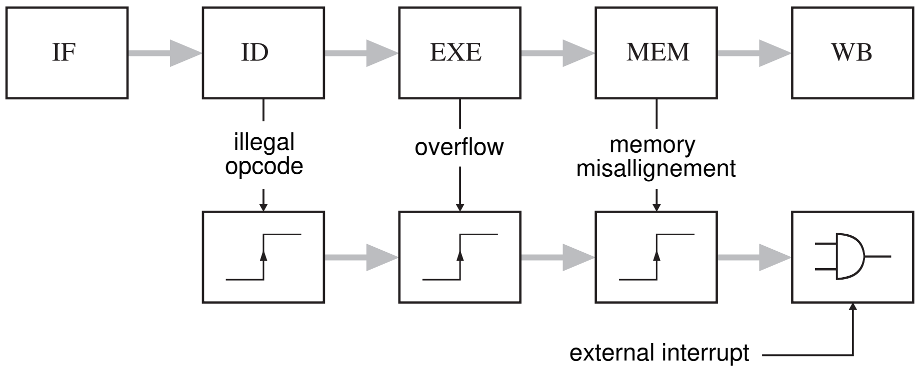

Figure 18. DLX Interrupt Pipeline

The high-level behavioral specification of the interrupt mechanism is a collection of properties related to the different tasks of the hardware at different time points of the pipeline execution. When an interrupts occurs, the information is transported via the interrupt pipeline till the WB stage where it is processed. A partial specification of the interrupt behavior is given as follows[8]:

$$
\begin{array}{l}
\vdash_{def} \mathit{INTR\_SPEC}\ (\mathit{PC, IAR, ill\_opc, overflow, mem\_mis, ext\_int}) := \\
\quad \forall\, t. \\
\qquad \ldots \\
\qquad (\mathit{PC}\ (t{+}5) = (\mathit{ill\_opc}\ (t{+}1)) \Rightarrow \mathit{C100} \mid \\
\qquad\qquad\qquad (\mathit{overflow}\ (t{+}2)) \Rightarrow \mathit{C200} \mid \\
\qquad\qquad\qquad (\mathit{mem\_mis}\ (t{+}3)) \Rightarrow \mathit{C300} \mid \\
\qquad\qquad\qquad (\mathit{ext\_int}\ (t{+}4)) \Rightarrow \mathit{C400} \mid \mathit{PC}\ (t{+}4) + 4) \wedge \\
\qquad \ldots \\
\qquad (\mathit{IAR}\ (t{+}5) = \mathit{PC}\ (t{+}4)) \\
\qquad \ldots
\end{array}
$$

where, the example of the correct setting of the program counter *PC* and the interrupt address registers (*IAR*) is specified. Here, the interrupt pipeline is checked during the WB stage of an instruction execution, jump to the appropriate vector addresses (represented by the constants *C100*, *C200*, *C300* and *C400*) is performed and the program counter *PC* is saved into *IAR*. Figure 19, for instance, shows the behavior of the program counter *PC* after an overflow happens during the EX stage. The predicate INTR_SPEC takes in addition to *PC* and *IAR*, the flags *ill_opc, overflow, mem_mis* and *ext_int* which are extracted from the interrupt pipeline at the *ID*, *EXE*, *MEM* and *WB*

8. A detailed formal specification and verification of *INTR_SPEC* for a hardware implementation of DLX is reported elsewhere [31].

stages, respectively. All temporal aspects are related to the fetch time t of a specific instruction. A quantification over t implies a quantification over all possible instructions fetched into the pipeline.

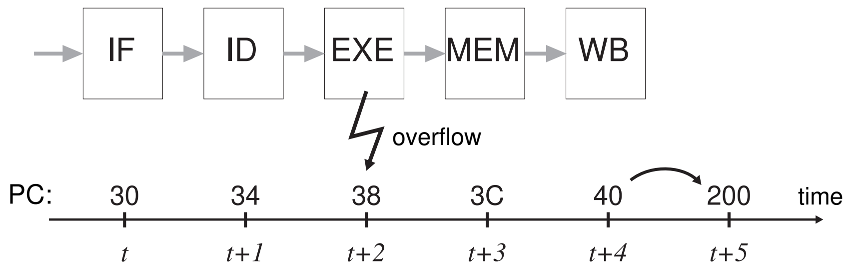

Figure 19. Behavior of the Program Counter *PC* by interrupts

The verification of the interrupt behavior cannot be done using the hierarchical model. The goal to verify

$$\vdash \; EBM \Rightarrow INTR\_SPEC$$

is split into subgoals related to the different aspects of the interrupt behavior as described above. These aspects (temporal properties) are then proven correct from the circuitries of the *EBM*. These circuitries are described in the *EBM* as block units with internal logic. For instance, the *EBM* includes several components at every pipeline stage building the interrupt pipeline. These blocks are available in the *EBM* specification as *INTR_ID*, *INTR_EXE*, *INTR_MEM* and *INTR_WB* and the *PC_UNIT* of the control unit of the hardware.

By proving

$$\begin{aligned}&\vdash \forall\, ill\_opc,\, overflow,\, mem\_mis,\, ext\_int.\\ &\quad INTR\_ID \wedge INTR\_EXE \wedge INTR\_MEM \wedge INTR\_WB \wedge PC\_UNIT \;\Rightarrow INTR\_SPEC(\ldots)\end{aligned}$$

we show that the interrupt properties are implied from the conjunction of the predicates of these units where they are expanded down to the gate level. Note that it will be quantified over every interrupt class in order to consider all possible combinations that should be executed correctly according to their priorities.

It is obvious from the above description that the verification of the interrupt behavior is a tedious task including many manual interactions. For the verification, we implemented a set of proof scripts based on an extensive use of arithmetic rewriting for the temporal aspects. The scripts used are tailored to the specific hardware implementation and cannot be transferred to other processors as is the case with the hierarchical pipeline data and control verifications.

10. Implementation in HOL

All formal specification and proof strategies of our methodology have been implemented using the higher-order logic theorem prover HOL [38] (version HOL90.6 which is based on SML [65]) within the MEPHISTO verification framework [56]. 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 HOL *tactics* and *tacticals* [38]. The HOL implementation of the model specifications, the temporal abstraction function and the pipeline data correctness is reported in [76]. The HOL implementation of the pipeline conflict formalization and verification process is reported in [78].

All implementations are kept general making them applicable to a wide range of simple RISC processors (e.g. as those used in embedded systems) and could be grouped as follows:

- implementations that need no instantiations and are directly usable for any RISC processor. These include the specification of pipeline conflicts (cf. sections 8.2.1, 8.3.1 and 8.4.1), the functions for predicate extractions (cf. section 8.1) and the proof tactics for pipeline control verification (cf. section 8.5)
- implementations that have to be parameterized according to the handled RISC processor. These are the temporal abstraction function (cf. section 5), the goal setting functions and the proof tactics for the pipeline data verification (cf. sections 7.1, 7.2 and 7.3)
- implementations for which general templates (illustrated by the DLX example) have been provided. These involve the specifications of the model levels (cf. section 6) and the type definitions for pipeline conflicts (cf. section 8.1)
- implementations for which no general patterns could be provided. These involve the specification and verification of specific hardware behaviors, e.g. interrupt, interlocks, branch prediction, etc. (cf. section 9) 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 [66], PVS [63], Nuprl [28], SDVS [57], LAMBDA [2], etc. is also possible.

11. Experimental Results on a VLSI Implementation of DLX

Using the already existing implementation of our methodology (cf. section 10), we made an interesting experimentation by letting a third person who implemented the processor in CADENCE performing the verification of this DLX. 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 [31]. 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 (see figure 12),
- definition of the arguments for the instructions and pipeline types and
- specification of the interrupt and bypass behaviors.

For formal correctness the following verification tasks were involved:

- verification of the pipeline data correctness,
- verification of the pipeline control correctness and

- verification of the interrupt and bypass behaviors.

With the exception of the bypass and interrupt behaviors the overall verification process has been achieved fully automatically.

During this experiment few design errors were found in the design which were, to our surprise, not discovered during the simulation process. Examples of these errors (bugs) are the implementation of a wrong addressing of the register file at the WB-stage for 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 pipeline data (class level) correctness proof and the second one during the pipeline control (data conflict) verification. Due to the hierarchical and constructive aspects of our methodology, these design errors 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 4 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 pipeline data correctness of the whole processor are given in table 5. Hereby it is interesting to notice that, as expected, the verification of the phase level corresponds to about 90% of the total pipeline data correctness proof overhead.

- The run times for the theorem generation of the *Used* and *Range/Domain* predicates are given in table 6.

- The run times for the pipeline conflict verification for the implemented DLX processor are given in table 7.

- The overall proof results including the verification of the interrupt and bypass behaviors are summarized in table 8. Accordingly, the whole verification of this DLX implementation took about one hour and required about seven million inferences.

Table 4. 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	-
Total	4515	182083	-

Table 5. Pipeline Data 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
Total	915.33	266945	-

Table 6. 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
Total	1862.49	3915216	-

Table 7. Pipeline Control 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
Total	2402.29	6754120	-

Table 8. 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
Pipeline Data Correctness	915.33	266945	95 main theorems
Pipeline Control Correctness	2402.29	6754120	3 main theorems
DLX Verification	3978.02	7044496	-

12. 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 pipeline data process of each single instruction and the correctness of the pipeline control during the 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 pipeline data correctness. Furthermore, we have shown that pipeline conflicts — resource, data and control conflicts — which occur in RISC cores can be conveniently modeled 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 constraints for software/hardware conflict resolution.

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 few parametrized 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. It is however restricted to single cycle execution and is not directly applicable to multiple instructions issue or multicycles function units as floating-point operations which are present in almost all recent superscalar commercial processors. While software based techniques as instruction scheduling is covered by the hierarchical verification approach, only a general template for the specification and verification of hardware based conflict resolution techniques has been provided within the scope of our methodology.

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. In [77] we have applied our methodology on the implemented DLX processor by handling the verification of the pipeline control correctness in a post-design manner in more or less a single step through all hierarchies.

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 multiple instruction issue, pipelined multicycles functional units, advanced dynamic scheduling and branch prediction techniques, 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. A. Berenbaum: *Functional Simulation for the CRISP Microprocessor*; Proc. International Conference on Very Large Scale Integration (VLSI87), Vancouver, Canada, August 1987, pp. 323-334.
9. A. Berenbaum; D. Ditzel; H. McLellan: *Introduction to the CRISP Microprocessor*; Proc. 32th IEEE Computer Society International Conference (COMPCON87), San Francisco, California, February 1987, pp. 86-90.
10. 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.
11. V. Bhagwati; S. Devadas: *Automatic Verification of Pipelined Microprocessors*; Proc. ACM/IEEE 31st Design Automation Conference (DAC94), San Diego, California, June1994, pp. 603-608.
12. M. Blomkvist; J. Nilsson; W. Sagefalk: *A VLSI Implementation of the DLX Microprocessor*; Department of Computer Engineering, Lund University, Sweden, September 1992.
13. 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.
14. A. Bode: *RISC-Architekturen*; BI-Wiss. Verlag, 1990.
15. R. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation*; IEEE Transactions on Computers, Vol. C-35, No. 8, August 1986, pp. 677-691.
16. 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.
17. O. Buckow: *Formale Spezifikation und (Teil-) Verifikation eines SPARC-kompatiblen Prozessors mit LAMBDA*; Diplomarbeit, Fachbereich Mathematik-Informatik, Universität-Gesamthochschule Paderborn, Germany, October 1992.
18. 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.
19. J. Burch*: Techniques for Verifying Superscalar Microprocessors*, Proc. 33rd ACM IEEE Design Automation Conference, Las Vegas, ACM Press, June 1996, pp. 552-557.
20. Cadence Design Systems Inc.: *CADENCE User Manuals*; Cadence Design Systems Inc., October 1991.
21. A. Camilleri: *Simulation as an Aid to Verification Using the HOL Theorem Prover*; Technical Report No. 150, Computer Laboratory, Cambridge University, October 1988.
22. CAO-VLSI Team: *Implementation of DLX in ALLIANCE*; MASI Laboratory, University Pierre et Marie Curie, Jussieu, Paris, France, March 1993.
23. P. Camurati; P. Prinetto: *Formal Verification of Hardware Correctness: Introduction and Survey of Current Research*; IEEE Computer, July 1988, pp. 8-19.
24. Coe, M.: Results from Verifying a Pipelined Microprocessor; Master Thesis, Laboratory for Applied Logic, University of Idaho, October 1994.

25. 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.
26. 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.
27. A. Cohn: *The Notion of Proof in Hardware Verification*; Journal of Automated Reasoning, Vol. 5, 1989, pp. 127-139.
28. R. Constable et al.: *Implementing Mathematics with the Nuprl Proof Development System*; Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
29. 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.
30. D. Cyrluk: *Microprocessor Verification in PVS: A Methodology and Simple Example*; Technical Report SRI-CSL-92-12, SRI Computer Science Laboratory, December 1993.
31. M. Dehof: *Formale Spezifikation und Verifikation des DLX-RISC-Prozessors*; Diplomarbeit, Institut für Technik der Informationsverarbeitung, Universität Karlsruhe, Germany, August 1994.
32. 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.
33. Digital Equipment Corp.: *Alpha Architecture Handbook*; Digital Equipment Corp., Maynard, Massachusetts, Order No. EC-H1689-10, 1992.
34. P. Dubey; M. Flynn: *Branch Strategies: Modelling and Optimization*; IEEE Transactions on Computer, Vol. 40, No. 10, October 1991, p. 1159-1167.
35. Electronic Design Interchange Format, *Version 2 0 0: EIA Interim Standard No. 44*; EDIF Steering Committee, Electronic Industries Association, May 1987.
36. S. Furber: *VLSI RISC Architecture and Organization*; Electrical Engineering and Electronics, Dekker, New York, 1989.
37. 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.
38. M. Gordon; T. Melham: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*; Cambridge, University Press, 1993.
39. A. Van De Goor: *Computer Architecture and Design*; Addison-Wesley, 1989.
40. 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.
41. M. Gordon: *Proving a Computer Correct using the LCF_LSM Hardware Verification System*; Technical Report No. 42, Computer Laboratory, University of Cambridge, September 1983.
42. B. Graham: *The SECD Microprocessor: A Verification Case Study*; Kluwer Academic Publishers, 1992.
43. A. Gupta: *Formal Hardware Verification Methods: A Survey*; Journal of Formal Methods in System Design, Vol. 1, No. 2/3, 1992, pp. 151-238.
44. A. Gupta; P. Stephan: *VHDL Design and Analysis of DLX*; CS252 Semester Project, University of California at Berkeley, May 1991.
45. 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.
46. 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.
47. J. Hennessy; D. Patterson: *Computer Architecture: A Quantitative Approach*; Morgan Kaufmann Publishers, Inc., San Mateo, California, 1996.
48. 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.
49. W. Hunt: *Microprocessor Design Verification*; Journal of Automated Reasoning, Vol. 5, No. 4, 1989, pp. 429-460.
50. W. Hwu; P. Chang: *Efficient Instruction Sequencing with Inline Target Insertion*; IEEE Transactions on Computer, Vol. 41, No. 12, December 1992, pp. 1537-1551.
51. Institute of Electrical and Electronics Engineers: *IEEE Standard VHDL Language Reference Manual*; IEEE Press, New York, June 1993.
52. Intel Corporation: *i860 64-Bit Microprocessor Programmer's Reference Manual*; Intel Corporation, Santa Clara, California, 1989.

53. J. Joyce: *Multi-Level Verification of Microprocessor-Based Systems*; PhD. Thesis, Computer Laboratory, Cambridge University, December 1989.
54. P. Kogge: *The Architecture of Pipelined Computers*; McGraw-Hill, 1981.
55. 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.
56. 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.
57. L. Marcus: *SDVS 10 Users' Manual*; Technical Report ATR-91(6778)-10, The Aerospace Corporation, 1991.
58. S. McFarling; J. Hennessy: *Reducing The Cost of Branches*; Proc. 13th Annual International Symposium on Computer Architecture, Tokyo, Japan, June 1986.
59. 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.
60. Mentor Graphics Inc.: *GENESIL Designer Manuals*; Mentor Graphics Inc., September 1989.
61. V. Milutinovic: *High Level Language Computer Architecture*; Computer Science Press, Inc., 1989.
62. Motorola, Inc.: MC88100 RISC Microprocessor User's Manual; Englewood Cliffs, New Jersey, Prince-Hall, 1988.
63. 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.
64. 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.
65. L. Paulson: *ML for the Working Programmer*; Cambridge University Press, 1991.
66. L. Paulson: *Isabelle: A Generic Theorem Prover*; Lecture Notes in Computer Science 828, Springer Verlag, 1994.
67. 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.
68. 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.
69. 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.
70. M. Srivas; M. Bickford: *Formal Verification of a Pipelined Microprocessor*; IEEE Software, Vol. 7, No.5, September 1990, pp. 52-64.
71. M. Srivas; S. Miller*: Applying Formal Verification to the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods*; Formal Methods in System Design, Kluwer Academic Publishers, Vol. 8, pp. 153-188.
72. H. Stone: *High-Performance Computer Architecture*; Addison-Wesley Publishing Company, 1990.
73. Sun Microsystems, Inc.: *The SPARC Architecture Manual*; Sun Microsystems, Inc., USA, Version 8, Part No. 800-1399-09, August 1989.
74. E. Talkhan; A. Ahmed; A. Salama: *Microprocessors Functional Testing*; IEEE Transactions on Computer Aided Design, Vol. 8, No. 3, March 1989.
75. 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.
76. 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.
77. 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.
78. 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.
79. M. Thomas: *The Industrial Use of Formal Methods*; Microprocessor and Microsystems, Vol. 17, No. 1, 1993, pp. 31-36.
80. N. Tredemick: *Experiences in Commercial VLSI Microprocessor Design*; Microprocessors and Microsystems, Vol. 12, No.8, October 1988.

81. 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.
82. P. Windley; M. L. Coe:*A Correctness Model for Pipelined Microprocessors*; In: R. Kumar and T. Kropf (Eds.), Proc. Theorem Provers in Circuit Design, Lecture Notes in Computer Science 901, Springer Verlag, 1995, pp. 33-51.
83. P. Windley*: Formal Modeling and Verification of Microprocessors*; IEEE Transactions on Computers, Vol. 44, No.1, 1995.
84. P. Windley*: Verifying Pipelined Microprocessors*; Proc. Conference on Hardware Description Languages (CHDL '95), Chiba, Japan, August 1995, pp. 503-511.
85. P. Windley, J. Burch: *Mechanically Checking a Lemma Used in an Automatic Verification Tool*; In: Srivas, M. and Camilleri, A. (Eds.), Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1166, Springer Verlag, 1996, pp. 262-276.
86. K. Winters: *ASIC Design Experience: MDLX*; Department of Electrical Engineering, Montana State University, USA, April 1992.
87. 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.