Higher Order Logic Theorem Proving and Its Applications
Joyce, J. and Seger, C. (Eds.)
Lecture Notes in Computer Science 780
Springer Verlag, 1994, pp. 281-294

Implementing a Methodology for Formally Verifying RISC Processors in HOL*

Sofiène Tahar[1] and Ramayya Kumar[2]

[1] University of Karlsruhe, Institute of Computer Design and Fault Tolerance (Prof. D. Schmid), P.O. Box 6980, 76128 Karlsruhe, Germany
e-mail:tahar@ira.uka.de

[2] Forschungszentrum Informatik, Department of Automation in Circuit Design, Haid-und-Neu Straße 10-14, 76131 Karlsruhe, Germany
e-mail:kumar@fzi.de

Abstract. In this paper a methodology for verifying RISC cores is presented. This methodology is based on a hierarchical model of interpreters. This model allows us to define formal specifications at each level of abstraction and successively prove the correctness between the neighbouring abstraction levels, so that the overall specification is correct with respect to its hardware implementation. The correctness proofs have been split into two steps so that the parallelism in the execution due to the pipelining of instructions, is accounted for. The first step shows that the instructions are correctly processed by the pipeline and the second step shows that the semantic of each instruction is correct. We have implemented the specification of the entire model and performed parts of the proofs in HOL.

1 Introduction

Completely automating the verification of general complex systems is practically impossible. Hence appropriate heuristics for specific classes of circuits such as finite state machines, arithmetic circuits, microprocessors, signal processors, etc., are needed for aiding the verification process.

Microprocessors build a particular class of well structured hierarchical circuits, that are used in a wide range of applications. Many microprocessor verification efforts have been done in the recent past; some using the HOL system [Cohn88,Joyc89,Wind90] and others based on functional calculi [Hunt87,SrBi90]. Most of them handle only one specific microprogrammed processor. The objective of our endeavour is the development of a generic methodology (comparable to that of Windley [Wind90]) for the formal verification of a large number of realistic RISC processor cores. Furthermore, we plan to integrate this methodology into a general verification framework [KuSK93]. In our previous work, we have developed and formalized a general hierarchical model for RISC cores and sketched the formal verification aspects using this model [TaKu93a,TaKu93b]. The aim of this paper is to show the practicability of our methodology by implementing it in HOL.

* This work has been partly financed by a german national grant, project Automated System Design, SFB No.358.

The organization of this paper is as follows. Section 2 presents briefly the hierarchical verification model. Section 3 describes the different abstractions used by the model with a focus on temporal abstraction. Section 4 gives a formalization of the abstract specifications and implementation in HOL. Section 5 discusses the verification steps and strategies adopted and includes HOL implementations of some proofs. Section 6 finally concludes the paper. Further, the presented methodology and implementation are exercised throughout this paper by means of a RISC example: DLX[1] [HePa90].

2 The Hierarchical Verification Model

In some related works, it has been shown that the verification of microprogrammed processors can be simplified by the use of a hierarchical model, called *interpreter model* [Joyc89,Wind90]. This model is based on abstraction levels which are introduced between the top-level specification and the circuit implementation EBM (Electronic Block Model). Structuring the verification of RISC processors using this interpreter model is not possible, since RISCs function differently. RISC instructions are executed in a number of physical steps, called pipeline stages (e.g. IF, ID, EX, WB) [HePa90], whose duration corresponds to one machine clock period. We define a *stage instruction* as the set of transfers, which occur during the corresponding pipeline segment. Each stage operation is partitioned into a number of sub-operations corresponding to the phases of a multiple phase non-overlapping clock. We define a *phase instruction* of a specific stage as the set of sub-transfers that occur during that clock phase. Usually, the RISC phase instructions are stage dependent and the stage instructions differ from one architectural instruction to another (table 1). These dependencies lead to a big number of verification steps between the EBM and the phase level. In order to circumvent this problem, we exploit the notion of instruction classes[2] and introduce the *class level* as the top level of our interpreter model. The stage and phase instructions are then parameterized in accordance with the class abstraction.

The overall verification model is given in Fig. 1, where the arrow between the levels means that the lower level is an abstraction of the next upper one. Using this hierarchical model, we are able to successively prove the correctness of all phase, stage and class instructions. The correctness of the architectural instructions can then be shown by a simple instantiation of the proofs for each particular instruction.

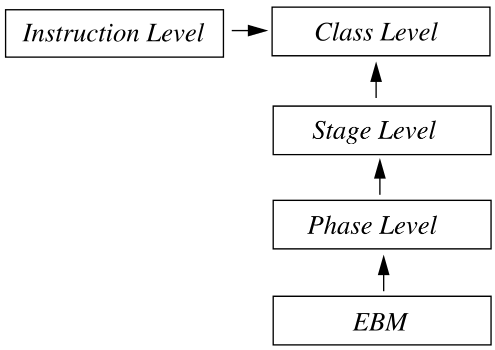

Fig. 1. RISC Interpreter Model

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

[2] An instruction class intuitively corresponds to the set of instructions with similar semantics, e.g. ALU, FLP, LOAD, CONTROL.

Table 1. DLX pipeline structure[3]

	ALU	LOAD	STORE	CONTROL
IF	IR $\leftarrow$ MEM[PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ MEM[PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ MEM[PC] PC $\leftarrow$ PC+4	IR $\leftarrow$ MEM[PC] PC $\leftarrow$ PC+4
ID	A $\xleftarrow[\phi_2]{}$ RF[rs1] B $\xleftarrow[\phi_2]{}$ RF[rs2] IR1 $\leftarrow$ IR	A $\xleftarrow[\phi_2]{}$ RF[rs1] B $\xleftarrow[\phi_2]{}$ RF[rs2] IR1 $\leftarrow$ IR	A $\xleftarrow[\phi_2]{}$ RF[rs1] B $\xleftarrow[\phi_2]{}$ RF[rs2] IR1 $\leftarrow$ IR	BTA $\xleftarrow[\phi_1]{}$ f (PC) PC $\xleftarrow[\phi_2]{}$ BTA
EX	ALUout $\leftarrow$ A op B or ALUout $\leftarrow$ A op (IR1)	DMAR $\leftarrow$ A+(IR1)	DMAR $\leftarrow$ A+(IR1) SMDR $\leftarrow$ B	
MEM	ALUout1 $\leftarrow$ ALUout	LMDR$\leftarrow$MEM[DMAR]	MEM[DMAR]$\leftarrow$SMDR	
WB	RF[rd] $\xleftarrow[\phi_1]{}$ ALUout1	RF[rd] $\xleftarrow[\phi_1]{}$ LMDR		

3 Abstraction Types

Within this verification model, we have different abstraction levels which have to be related to each other. It is important to consider a correct abstraction mechanism for the formal specification as well as for the formal verification. There are four kinds of abstractions [Melh88]: structural abstraction, behavioural abstraction, data abstraction and temporal abstraction. Structural and behavioural abstractions are a natural consequence of the hierarchical model. In the following we will briefly handle data abstraction and then we will focus on the concept of temporal abstraction.

3.1 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. The problem with using such concrete data representations is the presence of the implementation details of the data types. In [Joyc89,Wind90] uninterpreted data types were used. Here we use data types of a bit-vector theory. Since bit-vectors are naturally used in the description of the instruction set, we do not make any data abstractions and use bit-vectors through all abstraction levels. Thus, we save a lot of mapping functions between the concrete data type and an abstract one, e.g. natural numbers.

3.2 Temporal Abstraction

Temporal abstraction relates the different time granularities which occur in the formal specifications at various levels of abstractions [Melh88]. 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 (Fig. 2).

[3] In table 1, "$\leftarrow$" represents those stage transfers that are not broken down into phase transfers. "$\xleftarrow[\phi_1]{}$" ,"$\xleftarrow[\phi_2]{}$" represent the transfers which take place in phases 1 and 2, respectively.

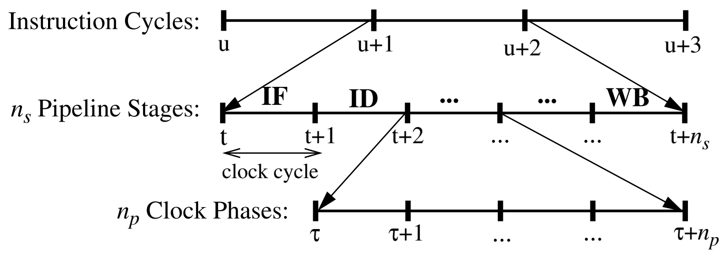

Fig. 2. Temporal abstraction

An ADD-instruction at the instruction level can be specified as follows:

ADD_SPEC (PC, I-MEM, RF, D-MEM):=

$$\forall \mathbf{u}\text{: } Inst\_cycle.\ RF(\mathbf{u+1})[rd(\mathbf{u})] = RF(\mathbf{u})[rs1(\mathbf{u})] + RF(\mathbf{u})[rs2(\mathbf{u})]^{4} \quad (1)$$

Referring to table 1, the ADD-instruction, using the more concrete time granularity, is:

ADD_IMP (PC, I-MEM, RF, D-MEM):=

$$\forall \mathbf{t}\text{: } Clk\_cycle.\ RF\,(\mathbf{t+5})[rd(\mathbf{t})] = RF(\mathbf{t+1})[rs1(\mathbf{t})] + RF(\mathbf{t+1})[rs2(\mathbf{t})] \quad (2)$$

A mapping function that relates abstract time scales in (1) to concrete ones in (2) is not linear since a unit of time at the abstract time scale does not necessarily correspond to one discrete time point of the next concrete time scale. For example, the same time point *u*, used in (1) for computing the addresses and reading the register file RF, have to be related to *t* and *t+1*, corresponding to the IF and ID-stage, respectively.

In a pipelined execution, state changes at the abstract level can take place at some time between the two discrete end-points of its time interval, i.e. at some time between *u* and *u+1* depending upon the implementation. A mapping time abstraction function should therefore take some implementation contexts into consideration while converting the abstract time to a more concrete one. A context parameter could be given as a tuple involving a read/write information and a pipeline stage identifier. For example, let *f* be the temporal abstraction function reading during the ID-stage yields -"*f ([read, ID], u) = t+1*"- and writing during the WB-stage yields -"*f ([write, WB], u+1) = t+5*"-.

Semantically, the affected state values in (1) have to be interpreted as follows: *X(u)* and *X(u+1)* correspond to the read and written values of *X* in the open intervals [u;u+1[and]u;u+1], respectively. Syntactically, these values of the specification must be supplemented with an abstraction function *f* in the verification goal, i.e. $(X \circ f)\ (u)$ instead of *X(u)*, where "∘" is the composition suffix (see Sect. 5.2).

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 time points within the clock cycle interval at some specific phases. In a similar manner, a time abstraction function has to be used in the appropriate verification goal. An implementation of a general time

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

abstraction function for both abstraction levels can be defined in HOL as follows:

```
val Time_abs = new_definition ("Time_abs",
      --`Time_abs n C u =
             let RW = (EL 0 C) in
             let ORD = (EL 1 C) in
                n*(u-RW) + ORD + RW `--);
```

The parameter n corresponds to the total number of implemented pipeline stages or clock phases. Assuming that stage and phase identifiers are ordered in some manner, the context parameter C is a tuple comprising of a read/write information and the ordinal value of the corresponding stage or phase identifier, e.g. $ORD\ (EX) = 2$. This definition of the time abstraction function assumes that written values are considered at the end of a discrete time interval while read values are those at the beginning.

This abstraction function has the advantage, that it allows specifications to be abstract and implementation independent. The various instantiations, i.e. n and C, are given later when making the appropriate verification goal (see Sect. 5.2).

4 Formal Specification

4.1 Instruction and Class Level

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

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

A Class instruction abstracts the semantic of a group of architectural instructions. The semantic of the ALU-class instruction, for example, can therefore be given as follows, where *op* abstracts all ALU-operations:

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

Let u be a unit of time for an instruction cycle. This ALU-instruction can be described formally in HOL by means of a predicate involving time as follows, where the function *BITS* extracts the bit-vector portion specified by the first two parameters:

```
val ALU_SPEC = new_definition ("ALU_SPEC",
      --`ALU_SPEC (PC,I_MEM,RF,D_MEM) op =
             let rs1 = BITS(25,21,(I_MEM PC)) in
             let rs2 = BITS(20,16,(I_MEM PC)) in
             let rd  = BITS(15,11,(I_MEM PC)) in
               !u. RF (u+1) (rd u) = op (RF u (rs1 u))
                                        (RF u (rs2 u))`--);
```

A first description of the implementation of the ALU-class, without structural abstraction and in terms of clock cycle units t can be given as follows:

```
val ALU_IMP = new_definition("ALU_IMP",
      --`ALU_IMP (PC,I_MEM,RF,D_MEM) op =
             let rs1 = BITS(25,21,(I_MEM PC)) in
             let rs2 = BITS(20,16,(I_MEM PC)) in
             let rd  = BITS(15,11,(I_MEM PC)) in
               !t. RF (t+5) (rd t) = op (RF (t+1) (rs1 t))
                                        (RF (t+1) (rs2 t))`--);
```

4.2 Stage Level

A stage instruction is defined as a set of elementary state transitions, that implement the corresponding semantic. Formally, a stage instruction is specified as a predicate on the visible states at this level. It is a conjunction of simple transfers that can be directly read off from the pipeline architecture (table 1) and encoded formally. For example, the ID-stage instruction common to the ID-row and the class columns ALU, LOAD and STORE is specified in HOL with the following predicate:

```
val ID_SPEC = new_definition ("ID_SPEC",
      --`ID_SPEC (A,B,RF,IR,IR1) =
             let rs1 = BITS(25,21,IR) in
             let rs2 = BITS(20,16,IR) in
               ! t. (A (t+1) = RF t (rs1 t)) /\
                    (B (t+1) = RF t (rs2 t)) /\
                    (IR1 (t+1) = IR t) `--);
```

An implementation of this stage instruction in terms of clock phases can be specified as follows:

```
val ID_IMP = new_definition ("ID_IMP",
      --`ID_IMP (A,B,RF,IR,IR1) =
             let rs1 = BITS(25,21,IR) in
             let rs2 = BITS(20,16,IR) in
               ! tau. (A (tau+2) = RF (tau+1) (rs1 tau)) /\
                      (B (tau+2) = RF (tau+1) (rs2 tau)) /\
                      (IR1 (tau+2) = IR tau) `--);
```

4.3 Phase Level

Phase instructions are also specified as conjunctions of elementary state transitions. These transfers are also read off directly from the pipeline structure (table 1) and encoded formally. State values of phase transitions that are not explicitly marked in the pipeline architecture, e.g. between IR and IR1-register in the ID-stage, are left unchanged until the last phase. From the ID-row and the class columns ALU, LOAD and STORE, the two phase instructions of the ID-stage are specified in HOL as follows:

```
val ID_Ph1 = new_definition ("ID_Ph1",
      --`ID_Ph1 (A,B,RF,IR,IR1) tau =
                IR (tau+1) = IR tau`--);

val ID_Ph2 = new_definition ("ID_Ph2",
      --`ID_Ph2 (A,B,RF,IR,IR1) tau =
             let rs1 = BITS(25,21,IR) in
             let rs2 = BITS(20,16,IR) in
                (A (tau+1) = RF tau (rs1 tau)) /\
                (B (tau+1) = RF tau (rs2 tau)) /\
                (IR1 (tau+1) = IR tau) `--);
```

4.4 Electronic Block Model

While the abstract levels of our interpreter model are behavioural descriptions, the EBM describes the structure of the hardware at the RT-level. However, the visible states and the temporal refinement used are the same as those of the phase level. At the top most level, the EBM is composed of the RISC processor and the interfaced instruction and data memory (cache). The processor consists conventionally of a datapath and a control unit, which are themselves compositions of simpler blocks (Fig. 3).

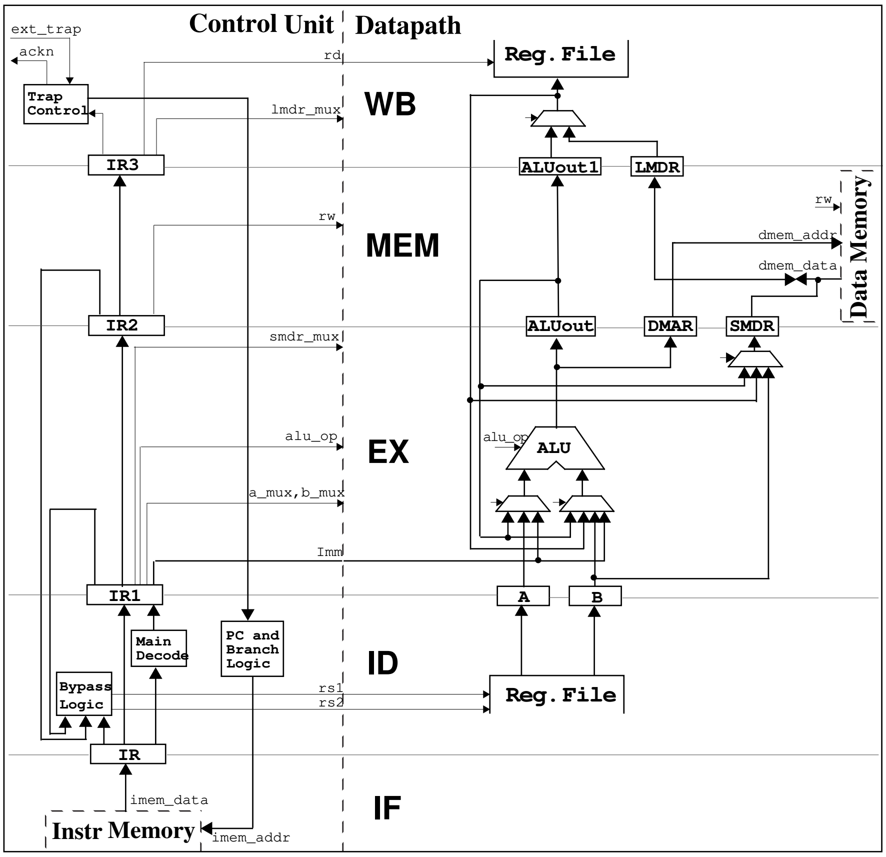

Fig. 3. Electronic Block Model of DLX (simplified)

Formally, the EBM is specified as a complex hierarchy of predicates, which are composed using conjunctions. The top level implementation of the EBM looks formally as follows, where the processor predicate is expanded into datapath and control unit:

```
EBM (PC,I_MEM,RF,D_MEM,A,B,ALUout,ALUout1,DMAR,SMDR,LMDR,
     IR,IR1,IR2,IR3,BTA,IAR,ext_trap,ackn,clk1,clk2) =
         ? 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,rw,ext_trap,ackn,
                      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)
```

Our approach is embedded within the MEPHISTO verification framework and the formal description of the circuit is obtained automatically from an EDIF output of a schematic representation, entered within a CAD tool [KuSK93]. The sub-blocks of the hierarchical design are broken into elementary library cells whose formal specifications are contained in a HOL-library.

5 Formal Verification

Potentially, a RISC processor executes n_s instructions in parallel, in n_s different pipeline stages (where n_s is the pipeline depth). These instructions could interfere with each other so that semantical inconsistencies could also occur. The formal verification of RISC processors consists therefore in showing that on one hand the semantics of the instruction set and on the other hand the pipelined sequencing of instructions are correctly implemented. Thus we split the correctness proof into two steps as follows:

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

$$SW\_Constraints, EBM \ \vdash \ Correct\_Instr\_Pipelining \tag{3}$$

2. assuming a correct pipelined sequencing, we prove that the EBM implements the semantic of each single architectural instruction correctly, i.e.:

$$\vdash \ EBM \Rightarrow Instruction\ Level \tag{4}$$

5.1 Correctness of Instruction Pipelining

The proof of correct instruction pipelining means to show that at any time, the parallel execution of instructions does not lead to any conflicts. There are three classes of conflicts, also called hazards, that can appear during the pipelined execution of any RISC machine [HePa90]:

- *structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions during simultaneous overlapped execution.
- *data hazards* arise when an instruction depends on the results of a previous instruction, i.e. data dependencies exist between them.
- *control hazards* arise from the pipelining of branches and other instructions that change the PC, i.e. interruption of the linear instruction flow.

The correctness of the pipelined execution of instructions is the direct consequence of the absence of all these hazards. We specify each hazard class formally as a predicate and define *Hazard_Pred* as their conjunction, i.e.:

$$Hazard\_Pred = Struct\_Pred \wedge Data\_Pred \wedge Control\_Pred$$

Thus, the correctness statement (3) remains in showing that the predicate *Hazard_Pred* is never true and the corresponding proof can be split as follows:

$$SW\_Constraints, EBM \ \vdash \ \neg Hazard\_Pred \tag{5}$$

$$\neg Hazard\_Pred \ \vdash \ Correct\_Instr\_Pipelining \tag{6}$$

The software constraints in (5) are those conditions which are to be met for designing the software so as to avoid conflicts, e.g. the number of delay slots to be introduced between the instructions while using a software scheduling technique. Further the EBM is assumed to include some conflict resolution hardware. The correctness of (6) includes the proof that all possible sequence combinations of n_s instructions are executed correctly.

In the following, we will sketch how hazards are specified and handled. The proofs of these steps are subject of ongoing work and are not reported here.

5.1.1 Handling Single Hazards

The different hazard predicates mentioned are closely related to our hierarchical interpreter model, since the data and control hazards are related only to the visible states at the instruction (class) level and the structural hazards are related to the stage and phase levels in a similar manner. For example, structural hazards at the stage level can be defined as follows:

$$\mathit{Stage\_Struct\_Pred} := \exists\, r{:}\ \mathit{Resource}.\ \exists\, S_i, S_j{:}\ \mathit{Stage\_Instruction}.$$
$$(\ \mathit{S\_type}(S_i) \neq \mathit{S\_type}(S_j)\ ) \wedge \mathit{S\_used\_Pred}(S_i, r) \wedge \mathit{S\_used\_Pred}(S_j, r)$$

where $\mathit{S\_type}(S_i)$ is a function which gives the logical type of the stage instruction S_i, e.g. ID or EX, and $\mathit{S\_used\_Pred}(S_i, r)$ means that the resource r, visible at the stage level, is used by the instruction S_i.

The formal specification of these hazards can be used for different purposes depending on the kind of verification applied:

- *on the fly verification*: the hazard predicates help the designer in synthesizing the hardware and software constraints needed for conflict resolution at a given step of the design process. For example, when a designer finds out that the predicate *Stage_Struct_Pred* is true for a given resource, that is used simultaneously by the class instructions i and j at stages S_n and S_m (Fig. 4), respectively, then either a software or a hardware constraint is extracted.
- *post verification*: the hazard predicates help the designer in validating some existing software or hardware constraints for conflict resolution.

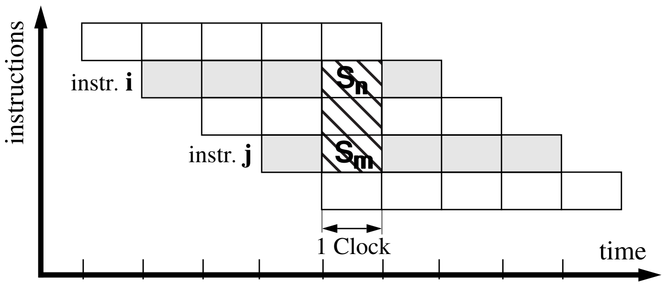

Fig. 4. Structural pipeline hazard

5.1.2 Handling Hazard Combinations

The proof of the predicate *Correct_Instr_Pipelining* ensures that all possible combinations of instructions that occur in the n_s pipeline stages are executed correctly. Exploiting the notion of class instructions, as described in Sect. 2, the case analysis explosion is avoided by considering the combinations of few classes.

In step (5), we have shown that structural, data and control hazards do not occur between any two instructions. In proving step (6), we have to show that for all possible permutations of the class instructions, the combinations of the instructions executing in the different pipeline stages do not lead to conflicts. This is done by splitting the proof into three steps, each corresponding to one kind of hazard. Each of these steps can then be proved by instantiating the theorems obtained in step (5), since they show the absence of conflicts between any two instructions in the pipeline.

5.2 Correctness of Single Instructions

Assuming the non existence of conflicts when instructions are executed in parallel within the pipeline, the correctness of each architectural instruction has to be shown. Using the notion of instruction classes and the hierarchical verification model, as described in Sect. 2, we first hierarchically prove the correctness of the class level, i.e.:

$$\mathit{EBM} \Rightarrow \mathit{Phase\ Level} \Rightarrow \mathit{Stage\ Level} \Rightarrow \mathit{Class\ Level.} \quad (8)$$

and then through instantiation, we show the correctness of the instruction level i.e.:

$$\mathit{EBM} \Rightarrow \mathit{Phase\ Level} \Rightarrow \mathit{Stage\ Level} \Rightarrow \mathit{Instruction\ Level.} \quad (9)$$

According to the abstract levels, each of these steps is broken into the following steps:

$$\mathit{Stage\ Level} \Rightarrow \mathit{Class\ (Instruction)\ Level,} \quad (10)$$

$$\mathit{Phase\ Level} \Rightarrow \mathit{Stage\ Level} \quad \text{and} \quad (11)$$

$$\mathit{EBM} \Rightarrow \mathit{Phase\ Level}\ . \quad (12)$$

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 a lot of general proof strategies could be developed. The goal settings and proofs are done automatically using generic functions and tactics. Further, each abstraction level can be verified independently, so that a designer is able to successively refine and verify the design.

In many aspects, this part of the verification process is similar to that used by Windley for the verification of microprogrammed microprocessors [Wind90]. In the following, we present the verification process at each level of abstraction and we then show how the instantiations are handled.

5.2.1 Stage-Class Level

Since the time granularities between the class and stage levels are different, the correctness proof is done in two steps:

Step I: We show that individual class implementations are correctly implemented by the cumulative effect of the corresponding stage instructions, i.e. for the ALU-class: $\mathit{IF\_SPEC} \wedge \mathit{ID\_SPEC} \wedge \ldots \wedge \mathit{WB\_SPEC} \Rightarrow \mathit{ALU\_IMP}$

Step II: This class implementation must now imply the abstract class specification, i.e.: $\mathit{ALU\_IMP} \Rightarrow \mathit{ALU\_SPEC}$

The specifications in step I are at the same time granularity (that of clock cycles). Hence, no temporal abstraction is needed in the verification goal. Using the formal descriptions of the class implementation and the stage specification, as described in sections 4.1 and 4.2, respectively, the verification goal and proof of this step for the ALU-class can be described in HOL as follows, where CLASS_IMP_TAC is a simple general tactic based mainly on several rewrites:

```
set_goal([],(--`!op.
                    IF_SPEC (PC,I_MEM,IR) /\
                    ID_SPEC (A,B,RF,IR,IR1) /\
                    EX_SPEC (A,B,ALUout,IR1,IR2) op /\
                    MEM_SPEC(ALUout1,ALUout,IR2,IR3) /\
                    WB_SPEC (RF,ALUout1,IR3)
                     ==> ALU_IMP (PC,I_MEM,RF,D_MEM) op`--));
e(CLASS_IMP_TAC(ALU_IMP,IF_SPEC,ID_SPEC,MEM_SPEC,EX_SPEC,WB_SPEC));
save_top_thm("ALU_IMP_CORRECT");
```

The specifications in step II use different time granularities. Therefore, the verification goal should include a temporal abstraction function and context parameters (as discussed in Sect. 3.2). The temporal abstraction function is instantiated with the corresponding pipeline depth n_s and is applied to the different context parameters. These implementation dependent context parameters are introduced for each state component in the specification formula. These parameters are existentially quantified and have to be instantiated appropriately later during the proof. For the proof, we use a general common tactic COMN_SPEC_TAC with the following parameters: the pipeline depth n_s, the pipeline stage identifiers, the corresponding class implementation and a list of the needed context parameters. An implementation in HOL of the COMN_SPEC_TAC tactic is given in following:

```
fun COMN_SPEC_TAC (depth, segments_list, imp, context_list) =
   MAP_EVERY EXISTS_TAC context_list THEN
   REPEAT (CONV_TAC let_CONV) THEN
   REWRITE_TAC [imp] THEN
   REPEAT STRIP_TAC THEN
   MAP_EVERY ASM_REWRITE_TAC
              [segments_list, [depth,R,W],
              [Time_abs,o_THM,EL,HD,TL,num_CONV(--`1`--)]] THEN
   BETA_TAC THEN
   CONV_TAC(DEPTH_CONV let_CONV) THEN
   ASM_REWRITE_TAC [ADD1,ADD_CLAUSES,ADD_SUB,SUB_0,
                    LEFT_ADD_DISTRIB,MULT_CLAUSES];
```

Using the specifications of the class level, as described in Sect. 4.1, and the time abstraction function, as described in Sect. 3.2, the verification goal and proof of step II for the ALU-class look formally in HOL as follows:

```
set_goal([],(--`
   !op.
      ?c1 c2 c3.
        let f = Time_abs ns in
            let rs1 = BITS(25,21,(I_MEM PC)) in
            let rs2 = BITS(20,16,(I_MEM PC)) in
            let rd  = BITS(15,11,(I_MEM PC)) in
               ALU_IMP (PC,I_MEM,RF,D_MEM) op
                 ==>  !u. (RF o f c1) (u+1) ((rd o f c3) u) =
                              op ((RF o f c2) u ((rs1 o f c3) u))
                                 ((RF o f c2) u ((rs2 o f c3) u))`--));
e (GEN_TAC THEN
   COMN_SPEC_TAC (ns, [IF,ID,EX,MEM,WB], ALU_IMP,
                 [(--`[W;WB]`--),(--`[R;ID]`--),(--`[R;IF]`--)]));
save_top_thm("ALU_SPEC_CORRECT");
```

The context parameters used in the proof are directly derived from the implemented pipeline architecture (table 1), e.g. *c2 = [R;ID]* corresponds to a *read*-operation during the *ID-stage*.

5.2.2 Phase-Stage Level

The correctness proof between the phase and the stage level is done in a manner similar to the previous section, by the two steps as follows:

Step I: proof that each stage instruction implementation is implied by the conjunction of the corresponding phase instructions, e.g. for the ID-stage:

$ID\_Ph1 \wedge ID\_Ph2 \Rightarrow ID\_IMP$

Step II: proof that the stage implementations imply the corresponding abstract stage specifications, e.g.: $ID\_IMP \Rightarrow ID\_SPEC$

Using the specifications of the stage and phase levels, as described in sections 4.2 and 4.3, respectively, the verification goal and proof of step I for the ID-stage look formally in HOL as follows, where STAGES_IMP_TAC is a simple general tactic based mainly on several rewrites:

```
set_goal([], (--`(!tau.  ID_Ph1 (A,B,RF,IR,IR1) tau /\
                         ID_Ph2 (A,B,RF,IR,IR1)(tau+1) )
                          ==> ID_IMP (A,B,RF,IR,IR1)`--));
e (STAGES_IMP_TAC (ID_IMP,ID_Ph1,ID_Ph2));
save_top_thm("ID_IMP_CORRECT");
```

Similar to the class level, the verification goal for step II involves temporal abstraction. The time abstraction function is instantiated with the corresponding number of clock phases n_p and context variables are included for each state component. The tactic COMN_SPEC_TAC is again used for the proof with the appropriate parameters: the number of clock phases n_p, the clock phase identifiers, the corresponding stage implementation and a list of the needed context parameters. These context parameters are directly derived from the implemented pipeline architecture (table 1).

5.2.3 EBM-Phase Level

The phase level lies directly above the EBM. This step of the verification is different from the previous proofs since the EBM is a structural specification, while the phase level is a behavioural one. However, due to the advantage of having used the hierarchical interpreter model, we only have to show the correctness of simple transfers. Although the specification of EBM is quite complex, a large amount of automation has been achieved in the domain of verification, e.g. MEPHISTO [KuSK93]. 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.

We have implemented a general tactic which proves all phase instructions automatically. This tactic, PHASES_IMP_TAC, is based on the tactics available in MEPHISTO. Using the specifications of the phase level and EBM, as described in sections 4.3 and 4.4, respectively, the verification goal and proof of the ID-phase2, for example, are given in HOL as follows:

```
set_goal([--` !t. (clk1 t = F) /\ (clk2 t = T)`--],
     (--`EBM (PC,I_MEM,RF,D_MEM,A,B,ALUout,ALUout1,DMAR,SMDR,
              LMDR,IR,IR1,IR2,IR3,BTA,IAR,ext_trap,ackn,clk1,clk2)
           ==> (!tau. ID_Ph2 (A,B,RF,IR,IR1) tau) `--) );
e PHASES_IMP_TAC;
save_top_thm("ID_PHASE2_CORRECT");
```

An implementation of PHASES_IMP_TAC is given in following, where EXPAND_DEFS_TAC, FLAT_HW_TAC and COMB_ELIM_TAC extracts all definitions of the current theory and its parent theories and expands the given goal, flattens a hierarchical description and eliminates combinatorial line variables, respectively:

```
val PHASES_IMP_TAC = EXPAND_DEFS_TAC THEN
                     FLAT_HW_TAC THEN
                     COMB_ELIM_TAC THEN
                     ASM_REWRITE_TAC[] THEN
                     REPEAT STRIP_TAC THEN
                     ASM_REWRITE_TAC[];
```

5.2.4 Instantiations

In this last part of the verification task, we deal with the correctness proofs of each instruction of the given architecture. This is obtained by simply instantiating the proofs at the class, stage and phase levels.

Since the class level is an abstraction of the instruction level (see Sect. 2), we first show the equivalence of particular instruction specifications to the instantiated related class specifications, e.g. for the ADD-Instruction we obtain the following theorem:

```
|- ADD_IMP (PC,I_MEM,RF,D_MEM) = ALU_IMP (PC,I_MEM,RF,D_MEM) add
```

Further, all correctness proofs at the class level have to be instantiated for the particular instructions of the instruction level. For example, we instantiate the proven theorem ALU_IMP_CORRECT of the ALU-class (cf. Sect. 5.2.1) with the operator constant *add* and obtain the following theorem for the ADD-instruction:

```
val ADD_IMP_CORRECT = SPEC (--`add`--) ALU_IMP_CORRECT;
val ADD_IMP_CORRECT =
  |- IF_SPEC (PC,I_MEM,IR) /\
     ID_SPEC (A,B,RF,IR,IR1) /\
     EX_SPEC (A,B,ALUout,IR1,IR2) add /\
     MEM_SPEC(ALUout1,ALUout,IR2,IR3) /\
     WB_SPEC (RF,ALUout1,IR3)
      ==> ALU_IMP (PC,I_MEM,RF,D_MEM) add : thm
```

From these two theorems, the correctness of the ADD-instruction is easily shown.

At the stage and phase levels, only the correctness of those instructions which include a class specific parameter has to be shown. The related theorems are obtained in a similar way as for the class level by simple instantiations.

6 Conclusions and Future Work

In this paper, we have presented a partly implemented methodology for the verification of RISC cores in HOL. This methodology is based on a hierarchical interpreter model which closely reflects the design hierarchy used for designing RISCs. 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 in HOL. Further, the verification process could be ensured by splitting the proof goal into two parts, namely the correctness of the pipelined execution of various instructions and the correct implementation of the semantic of each single instruction by the hardware. The ease of formalizing the specifications at each level of abstraction and the similarity of the proofs between the levels have been illustrated by some simple examples implemented in HOL. The methodology is generic, in that it provides 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.

The implementations have been done in HOL90 within the MEPHISTO verification framework. We have implemented different specifications and proof strategies at each level of abstraction. Proof strategies for the multiple conflict cases are at present under development. We have also implemented the hardware for the DLX in a commercial VLSI design environment and are using it as one of the benchmarks for our methodology.

References

[Cohn88] Cohn, A.: A Proof of the Viper Microprocessor: The First Level; In: VLSI Specification, Verification and Synthesis, Eds. G. Birtwistle and P.A. Subrahmanyam, Kluwer, 1988.

[HePa90] Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach; Morgan Kaufmann Publishers, Inc. San Mateo, California, 1990.

[Hunt87] Hunt, W.: The Mechanical Verification of a Microprocessor Design; In: From HDL Description to Guaranteed Correct Circuit Designs, Ed. D. Borrione, North-Holland, 1987.

[Joyc89] Joyce, J.: Multi-Level Verification of Microprocessor-Based Systems; PhD thesis, Cambridge University, December 1989.

[KuSK93] Kumar, R., Schneider, K., Kropf, Th.: Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment; Journal of Formal Methods in System Design, Vol. 2, pp. 165-230, 1993.

[Melh88] Melham, Th.:Abstraction Mechanisms for Hardware Verification; In: VLSI Specification, Verification and Synthesis, Eds. G. Birtwistle and P. A. Subrahmanyam, Kluwer, 1988.

[SrBi90] Srivas, M., Bickford, M.: Verification of a Pipelined Microprocessor Using Clio; In: Hardware Specification, Verification and Synthesis: Mathematical Aspects, Eds. M. Leeser and G. Brown, Springer, 1990.

[TaKu93a] Tahar, S., Kumar, R.: A Formalization of a Hierarchical Model for RISC Processors; to appear in Proc. of Euro-ARCH'93, Munich, Germany, Springer Verlag, 1993.

[TaKu93b] Tahar, S., Kumar, R.: Towards a Methodology for the Formal Hierarchical Verification of RISC Processors; to appear in Proc. of the 1993 International Conference on Computer Design, Cambridge, Massachusetts, IEEE, 1993.

[Wind90] Windley, P.: The Formal Verification of Generic Interpreters; PhD thesis, University of California, Davis, Division of Computer Science, July 1990.