Verification of Pipelined Microprocessors Using Invariants

Mustapha Bourahla

University of Biskra,
Computer Science Department,
BP. 145 RP, Biskra, Algeria, 07000,

Email: mbourahla@hotmail.com

Mohamed Benmohamed

University of Constantine,
Computer Science Department,
Constantine, Algeria, 25000,

Email: ibnm@yahoo.fr

Abstract: *This paper presents a new approach for the verification of a pipelined microprocessor which is based on the definition of invariants to characterize the reachable states of the pipelined machine. To express many machine-relevant properties, we have modelled the stream of instructions with the system Maude which is based on Rewriting Logic. It is also used to run and debug the pipelined machine specification. The meta-level module ITP (Inductive Theorem Prover) is used to verify the pipelined machine properties, presented as its object level specification, and eventually to verify a complete pipelined machine design, whose correctness is defined using the idea of pipeline flushing.*

Keywords: *Formal Verification, Rewriting Logic, Formal Specification, Pipelined Microprocessors.*

Received: *July 16, 2003* | ***Revised:*** *February 15, 2004* | ***Accepted:*** *March 11, 2004*

1. Introduction

Design errors in pipelining which is a key feature of today's microprocessor designs [10], can often lead to violations of the sequential semantics. For example, an update to a register or memory location by one instruction may not be detected by an instruction following too closely in the pipeline. An instruction following a conditional branch may be executed prematurely, modifying a register even though the processor later determines that the branch is taken. Such hazard possibilities increase dramatically as the instruction pipelines increase in both depth and width.

Validation by simulation becomes increasingly costly and unreliable as processors increase in complexity. As an alternative to simulation, a number of researchers have investigated using formal verification techniques [7, 8]. There have been several earlier successes to formally verify pipelined machines. Burch and Dill [4] were the first to demonstrate that automated decision procedures for a logic of equality with uninterpreted functions (EUF) could be used to verify pipelined processors. They assume there are two abstract models of the processor; a "program" model providing a direct implementation of the instruction set, and a "pipeline" model that captures the complexities of the actual implementation. Verifying that the pipelined processor has behaviour matching that of the program model can be performed by constructing (using symbolic simulation) a formula in EUF that compares for equality the terms describing the modifications to the programmer-visible state (i.e., the registers, data memory, and program counter) produced by the two models and then proving the validity of this formula.

We claim two problems with this approach: First, the implementation (pipeline model) should be symbolically simulated several times to model the effect of flushing the pipeline which means case explosion. The second problem is that the pipeline machine is not valid for all the states but just the reachable states. Burch and Dill suggested that they can use invariants to restrict the "old pipeline state", but they didn't say how to do that or how to express them. The proofs require invariants, a logical formula characterizing a superset of the states reachable from the initial state of the microprocessor.

Our method introduces a new technique to express the invariants and to prove their closure property. This method is based on the specification of the instructions stream in the pipeline (we call it pipeline state) using the algebraic specification language Maude [5], which allows us to directly express many machine-relevant properties. Using this representation, we can verify pipeline properties incrementally, and eventually verified a complete pipelined machine design, whose correctness is defined using the idea of pipeline flushing. The proof has been checked by the ITP (Inductive Theorem Prover) [6]. The ITP is a meta-level module of the system Maude for proving theorems.

The rest of the paper is organized as follows: a design of a pipelined machine to be the base of our verification technique is given in Section 2. In Section 3, we discuss how to represent the correctness of the pipelined machine and why it is difficult to verify. In Section 4, we present the specification of an abstracted

pipelined machine. Section 5 presents the proof of our verification. Some concluding remarks are given in Section 6.

2. Design of a Pipelined Machine

To investigate verification techniques for today's complex pipelined processors, we have designed a simple five-stage pipelined machine based on the DLX architecture [10]. The small number of chosen instructions is not an issue since, as we show later, the verification would not change significantly with the addition of new instructions. The instruction set contains instructions from most of the important classes of instructions one would find in any instruction set: ALU instructions, immediate instructions, branch instructions, jump instructions, load instructions, and store instructions.

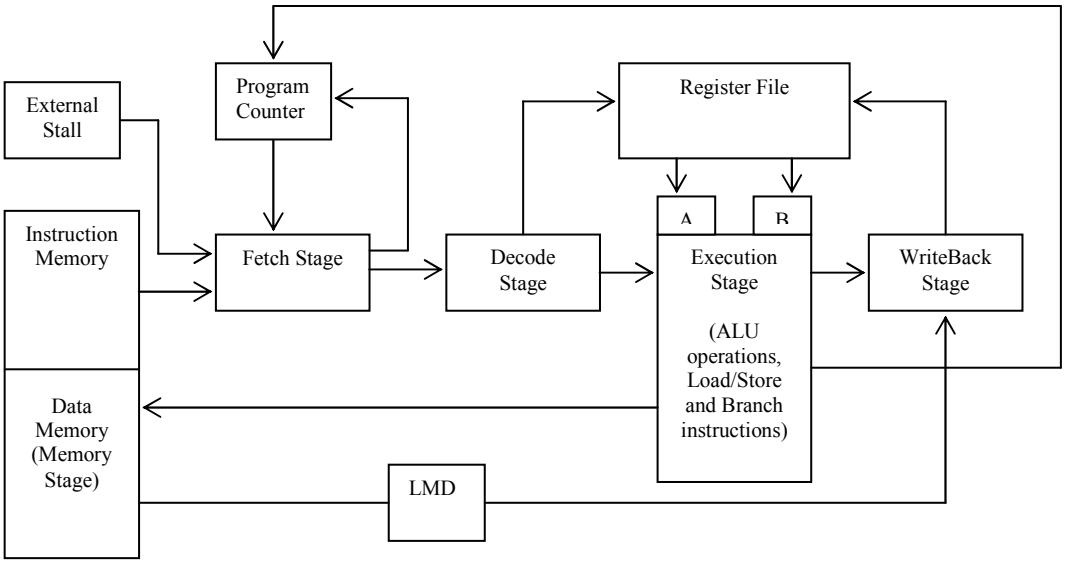

Figure 1: A simple five-stage pipelined machine

The organization of the processor is shown in Figure 1. The register file is written by the write-back unit, and read by the decoding unit. The register file has two read ports to support simultaneous reading of multiple source operands, but has only one write port. The memory is accessed only by the instruction-fetch unit and the execution unit. The memory is divided into two parts: the last half is readable and writable, while the first half is read-only. Instructions can be fetched only from the first half memory; this restriction eliminates the problem of self-modifying code as long as code is fetched from the first half of memory. The memory always responses in one cycle. Also, simultaneous accesses by the instruction fetch unit and the execution unit are allowed, as is often the case with a split cache. The pipeline is split into stages. They are named IF (Instruction Fetch) stage, DC (Decode) stage, EX (Execution) stage, MEM (Memory) stage and WB (Write Back) stage. At the beginning of the processing of each instruction, the instruction fetch unit reads the instruction pointed to by the PC (program counter) from the memory and sends it to the decode unit. In the presence of a stall event the content of the fetch unit will not be changed. During the decode stage, the contents of the source registers of the instruction in the fetch stage are retrieved from the register file and stored into the ALU latches A and B (the immediate data are retrieved directly from the instruction). The decoded instruction will be send to the execution unit. Before sending the decoded instruction, a check should be done for read after write dependencies between the content of the instruction register of the decode stage and first, with the instruction currently in execution stage, and then with the instruction in the memory stage. We assume that the write operation to the register file is always in the first half of a clock cycle and the read operation is always in its second half. If there is such a read after write dependency, the instruction is suspended until the appropriate data is ready. The design can use a bypassing logic to reduce the number of suspensions [10]. There are two major operations in the execute stage: the appropriate operation (ALU operation or Load/Store operation) is performed first, the second operation is the calculation of the new value for the program counter. The memory stage calculates values for the load memory data register (LMD) if the current instruction is a load or the stores a value in the data memory if the current instruction is store. The write-back stage updates the register file if necessary.

3. Pipeline Correctness Diagram

Our research objective is proving the equivalence between a sequential specification (Sequential) and pipelined design (Pipelined); however, this equivalence is hard to define. Intuitively, we want to verify that the pipelined design executes all instructions in any context just as the sequential machine does. A central problem is how to define the effect of a single instruction in the pipelined machine. In a pipelined design, the machine starts the execution of an instruction before an earlier instruction completes, blurring the execution boundary between instructions. On the other hand, a sequential machine completes each instruction before starting the next, making the boundary between instructions clear. Suppose that the contents of memory are as follows.

I_0: add r_b r_a r_b
I_1: add r_a r_b r_a

When this simple two-line code fragment is executed on the (Sequential) and (Pipelined) machines, we get the following traces.

Clock	Sequential	Pipelined	I_0	I_1
0	<0, <1,1>>	<0, <1, 1>>		
1	<1,<1,2>>	<1, <1,1>>	IF	
2	<2,<3,2>>	<2,<1,1>>	DC	IF
3		<2,<1,1>>	EX	Stall
4		<2, <1,1>>	MEM	Stall
5		<3,<1,2>>	WB	DC
6		<4, <1,2>>		EX
7		<5, <1,2>>		MEM
8		<6, <3,2>>		WB

Table 1: Execution traces in the sequential and pipelined machines

The rows correspond to steps of the machines, e.g., row Clock 0 corresponds to the initial state, Clock 1 to the next state, and so on. The Sequential and Pipelined columns contain the relevant parts of the state of the machines: a pair consisting of the PC and the register file (itself a pair consisting of registers R_a and R_b). The final two columns indicate what stage the instructions are in (only applicable to the pipelined machine). In the initial state (in row Clock 0) the PCs of the sequential and the pipelined machines contain the value 0 (indicating that the next instruction to execute is I_0) and both registers have the value 1. In the next sequential state (in row Clock 1), the PC is incremented and the add instruction performed, i.e., register R_b is updated with the value $R_a + R_a = 2$. The final entry in the sequential column contains the state of the sequential machine after executing I_1. After one step of the pipelined machine, I_0 completes the fetch phase and the PC is incremented to point to the next instruction. After step 2 (in row Clock 2), I_0 completes the decode stage, I_1 completes the fetch phase, and the PC is incremented. After step 3, I_0 completes the execution phase but I_1 is stalled in this step because one of its source registers is R_b, the target register of the previous instruction (normally, the bypassing logic sends the execution result to the ALU latches which eliminates this stall). After the memory stage, at step 5, I_0 completes the write-back phase and the register file is updated for the first time with R_b set to 2. Since the previous instruction has not completed, the value of R_b is not available and I_1 is stalled for another cycle (the bypassing logic will eliminate also this stall). In the next cycle, I_1 enters the decode stage and I_2 enters the fetch stage (not shown). In step 6, I_1 is in the execution stage and then it will enter the memory stage. Finally, I_1 is completed in step 8 (write-back stage) and register R_a is updated.

A more general approach for comparing a pipelined design with a sequential specification was proposed by Burch and Dill [4]. Figure 2 shows the basic idea of such a comparison. We map a pipelined machine state to a sequential machine state by flushing instructions out of the pipeline without fetching any new instructions, and then projecting the flushed pipeline state to a sequential machine state. Using this mapping, we can define a correctness criterion for our pipelined machine by the diagram shown in Figure 2.

Where $F_{Pipe}()$ and $F_{Seq}()$ are the transition functions for the pipelined and sequential machines. $F_{Pipe}(., I_{Stall})$ will give the next pipeline state by applying an external stall event. By contrast, $F_{Pipe}(., I)$ will give the next pipeline state without a stall event (which means fetching the new instruction I). The function *proj()* will project the flushed pipeline state to its equivalent sequential state. By the same way, $F_{Seq}(., I)$ will give the next sequential state for the instruction I. For example (see the table above), if the old pipeline state is <1, <1,1>> after its flushing we get the pipeline state <1, <1,2>>. By mapping of this flushed pipeline state to sequential state, we will have the sequential state <1, <1,2>>. The new pipeline state of <1,<1,1>> is <2, <1,1>>, the state after its flushing is <2, <3,2>> and its projection is the sequential state <2,<3,2>>.

This diagram holds only when the pipelined machine fetches an instruction during the left-hand side pipeline state transition, so we have to prove a similar diagram for the case when no instruction is fetched. Once the diagrams for these two cases are proved, we can prove an n-cycles version of the diagram, which implies that the state transitions of the sequential machine can be emulated by the pipelined machine as long as the initial states of the pipeline state transitions are flushed.

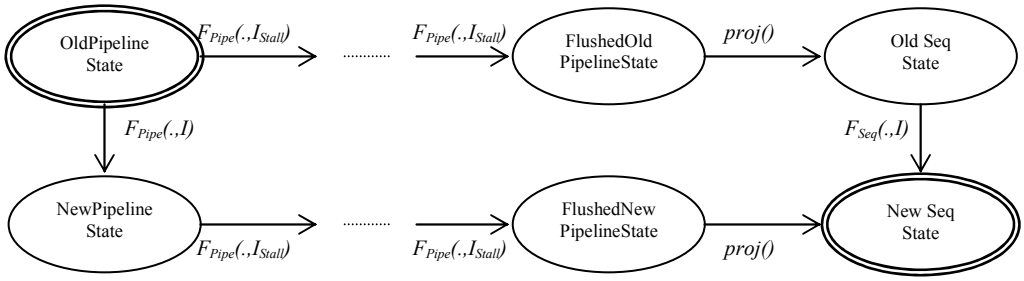

Figure 2: Correctness diagram

In the original work by Burch and Dill, they proved the pipeline correctness diagram by symbolically simulating the pipelined machine design in their logic of uninterpreted functions with equalities. Just like our example, their behavioural machine specifications are given by next state functions. By unrolling these functional specifications, they compare the traversal of the paths in Figure 2. The comparison is done for any initial pipeline configuration by an automatic decision procedure with case analysis and simplification. This method is much faster than the ordinary simulation, because only the control part of the pipelined machines is analysed, while the data path is manipulated symbolically. One problem of the symbolic execution is the explosion in the number of examined cases when the pipeline-control logic becomes complex.

Because the pipeline correctness diagram involves flushing which takes many machine cycles. The second problem is that the diagram is not valid for all pipeline states, but just the reachable states. For example, if an invalid instruction is in the pipeline, it will not give a next state which makes such verification not working. Burch and Dill suggested that they can use invariants to restrict the "old pipeline state". However, it is not clear from the paper how to prove the closure property of the invariants during the pipeline state transition, or even how to express such invariants. We will use the pipeline correctness diagram as our verification goal, because it is the best and most general method we know of to represent the correctness of a complex pipelined machine. In the following two sections, we discuss our method to prove the correctness diagram and our approach for the problems discussed above.

4. Abstraction of Pipelined Machine

Our approach is based on analytical reasoning methods using a theorem prover, rather than testing the control logic for all possible pipeline states with an automatic decision procedure. By verifying the machine incrementally at an abstract level, we aim at reducing the computational cost of verification.

To specify our pipelined machine, we have used the syntax of the engine Maude which is a Rewriting Logic system implementing the algebraic specification and it extends the equational logic [5]. The version of equational logic that provides the logical basis for functional modules is membership equational logic. Membership equational logic is an expressive version of equational logic supporting sorts, subsorts, partiality, and overloading of functions symbols.

Maude functional modules are assumed to be confluent and terminating, and their operational semantics is equational simplification, that is, rewriting of terms until a normal form is obtained. It was also necessary to write a specification of the sequential machine for mapping during verification. These two specifications are executable which allows their debugging. The abstraction of the specified pipelined machine is the specification of another high level to express the pipeline configuration composed from the stream of instructions and status of the important components (program counter, register file and data memory).

These specifications should also be accepted syntactically by the ITP (Inductive Theorem Prover) tool which is a meta-level module of Maude for proving properties of functional modules, i.e., specifications in membership equational logic (a subset of the Rewriting Logic) with an initial algebra semantics [6]. The ITP tool implements a sound inference system for proving properties of functional modules. In this paper, we do not present the Rewriting Logic and the two systems Maude and ITP. Our concentration is on their application to the specification and the verification of the pipelined microprocessor. Therefore, the rest of the paper is written by a fashion to help the reader not familiar with the Rewriting Logic.

The stream of instructions in the pipeline, is modelled as an union of instruction states (we call it pipeline state) each instruction state represents the state of an instruction being executed. This representation is devised so that we can directly reason about the effect of instructions as they flow through the pipeline. This pipeline state represents all the instructions which are

currently being executed in the pipeline. The order of the instructions in this expression is the same as the order that instructions are processed by the sequential specification. The following operator represents the specification of an instruction state which is a set expressing necessary information on the execution of an instruction:

```
op _:_ : ComponentLabel ComponentState ->
       Component .
op _;_ : ComponentList ComponentList ->
       ComponentList .
op <_> : ComponentList -> InstructionState .
```

An instruction state is composed of a list of components; each component is specified by a label and a state value. The labels are defined as operators.

```
ops Inst PC Stg A B Imm AluOut Cond LMD : ->
       ComponentLabel .
```

Where *Inst* is for the instruction definition, *PC* defining the program counter in the logical sequential state for this instruction and *Stg* defining the stage at which this instruction is in, the retrieved instruction arguments are specified by the labels *A*, *B* and *Imm* and the result of the execution will be the value of the label *AluOut*. The label *Cond* will specify the value of a conditional expression for branch instruction and *LMD* is the label of the register Load Memory Data to contain the data to be loaded to the register indicated by the current instruction, of the register file. The instructions are: store word, load word, unconditional jump, conditional branch (branch when the source register is equal/not equal to zero), 3-register ALU instructions, and ALU immediate instructions. The specifics of the ALU operations are abstracted away in both the specification (sequential machine) and the implementation (pipelined machine). Thus, our verification covers any set of ALU operations, assuming that the combinational ALU in the processor has been separately verified.

```
op STORE : Register Address -> Instruction .
op LOAD : Register Address -> Instruction .
op LOAD-IMM : Register Data -> Instruction .
op JUMP : Address -> Instruction .
op BEQZ : Register Address -> Instruction .
op BENZ : Register Address -> Instruction .
op ALU-OP : Register Register Register ->
       Instruction .
op ALU-IMM : Register Register Data ->
       Instruction .
op _(_) : Data Register -> Address .
```

For example, an instruction *add* in the decode stage will be represented by the following term of sort *InstructionState*: *<Inst: ALU-OP ra rb ra; PC: 4; Stg: DC; A: 5; B: 2; Imm: 0>*. The stream of instructions is an union expression of all the values of the instruction states which is a sort of *PipelineState*. The identity of *PipelineState* is a term of sort *InstructionState*. The empty configuration has the value *nil*. The following union operator -- expresses a multi-set which specifies the pipeline state.

```
subsort InstructionState < PipelineState .
op nil : -> PipelineState .
op -- : PipelineState PipelineState ->
       PipelineState .
```

The pipeline design state has the defined sort *PipelineDesignState* which is the combination *<PipeState:_; PC:_; regfile:_; dmem:_>*. Where the component *PipeState* is of the sort *PipelineState*. It is obvious from this presentation that the execution of the specification of the pipelined microprocessor will yield a trace represented by a term of sort *PipelineDesignState*. This term will be used for verification. One important think to verify is if this term is a correct representation of the pipelined machine state. It is necessary to define the following transition function `NewPipelineState()`. The second argument is to allow external stalling or not.

```
op NewPipelineState : PipelineDesignState
       Bool ->  PipelineDesignState .
```

Every reachable pipelined machine state and its correct term representation should satisfy the following properties:

- ***NoStageConflict*** : No instructions in the pipeline state representation should share the same pipeline stage.
- ***ValidInstructionStrea****m*: The logical sequential states satisfy the defining equation: $(pc_{i+1}, regs_{i+1}, mem_{i+1}) = F_{Seq}(pc_i, regs_i, dmem_i)$.
- ***NoRAWHazard****s*: Non-existence of read after write hazards.
- ***PCCorrespondenc****e*: the program counter in the pipeline machine correctly points to the next instruction to be executed.

We define the predicate *isCorrect()* to be the conjunction of the pipeline properties listed above. If *ps* is a correct representation of a reachable pipelined machine state, the predicate *isCorrect(ps)* should hold which means that all the properties are satisfied. In order to prove that the properties hold for all the reachable machine states, we need to show that these properties are invariant during the pipeline state transition. If the term *ps* correctly represents the pipeline design state, the new term after an update should also be the correct representation of the new pipelined state after a machine clock cycle.

Lemma 1 (Invariant property of isCorrect()) *If isCoorect(ps) holds then isCorrect(NewPipelineState (ps)) holds.*

The empty instruction stream *"nil"* is the correct representation of a flushed pipeline machine state, that is, no partially executed instruction is in the pipeline of this state, and all the properties hold.

Lemma 2 (Representation for a flushed State) *Suppose ps is a flushed state, isCorrect(<PipeState: nil; PC: pc; regfile: rf; dmem: dm>) holds.*

By Lemma 1 and Lemma 2, we can conclude that for any pipeline state which is reachable from a flushed pipeline state, there is a representation *ps* satisfying all the properties which means *isCorrect(ps)* holds. This fact will be used in the proof of our correctness diagram in the next section.

5. Proof of the Pipeline Correctness Diagram

Before proceeding to the proof of the pipeline correctness diagram, we first have to specify the pipeline flushing process. Pipeline stalling is a newly defined behaviour of the pipeline in which the machine does not fetch a new instruction, but works normally otherwise. We define *Stall* as the operation specifying one clock cycle of stalling. Then, the entire pipeline flushing process is defined as a recursive operation *Flus*h, which repeatedly calls Stall until all the instructions are flushed out of the pipeline.

```
op Stall : PipelineDesignState ->
       PipelineDesignState .
op Flush : PipelineDesignState ->
       PipelineDesignState .
```

Lemma 3 (Invariant Property in Stalling) *If ps is a correct representation of a pipeline state, the updated representation for the pipeline state Stall(ps) after one cycle of stalling should be also correct. Which means isCorrect(Stall(ps)) holds if isCorrect(ps) holds.*

Lemma 4 (Termination of flushing) *For any pipeline state, if there exists a correct representation ps, then the recursively defined function Flush(ps) terminates and returns a flushed pipeline state.*

Lemma 5 (Flushing of Flushed Pipeline) *For any flushed pipeline state ps, Flush(ps) = ps.*

The proof of the diagram in Figure 2 has to show that the same "new sequential state" can be obtained from any configuration of the "old pipeline state" by following the two paths in the diagram.

Theorem 1 (Pipeline Correctness Diagram) *If a pipelined machine state and its representation ps satisfy the properties defining the invariant, and the pipeline machine fetches a new instruction in the next clock cycle, then:*

(goal fmod PIPELINE is…endfm |- {ps} ((isCorrect(ps)) =>
((NewSeqState(proj(flush(ps)))) ==
(proj(flush(NewPipelineState(ps)))))) .)

The above theorem is written with respect to the ITP syntax. Where *fmod PIPLINE is…endfm* is the functional Maude module specifying the pipelined machine (a piece of its code is presented as appendix of this paper), and *proj()* is the projection function from pipeline states to sequential states. The predicate *isCorrect()* and the transition function *NewSeqState()* are also defined in this module.

The pipeline correctness diagram may not hold for some unreachable states. In Theorem 1, we constrain "old pipeline state" by assuming the existence of a correct representation *ps* which satisfies *isCorrect(ps)*. This excludes some unreachable states from "old pipeline states", but Theorem 1 still covers all the reachable pipeline states. This is justified by Lemma 1 and Lemma 2.

The proof of Theorem 1 is split into separate proofs about the program counter, the register file, and the data memory. Our method of diagram proof using the algebraic representation of the pipeline configuration solves the problem for the symbolic execution method discussed in Section 3. It does not suffer the case explosion problem, because we logically proved the correctness of the diagram at an abstract level using various pipeline properties. And our method excludes the inconsistent pipeline states from the "old pipeline state" by assuming the existence of a correct representation for the pipeline state.

The execution capability in Maude allows us to run the behavioural description of our pipelined machine. Using this feature, we have run small programs on our machine design and discovered most of our design errors before applying formal verification techniques. Actually, all the fatal errors were eliminated in this stage. We believe it is important to remove most of the design errors using techniques like simulation before attempting to apply formal verification techniques.

The actual verification work did not proceed quite as directly as we described. We first proved the pipeline correctness diagram under the assumption that Lemma 1 holds, then we proceeded to the proof of Lemma 1, which took most of our verification effort. During this proof, we often discovered that we needed to prove

additional properties about the pipeline and instructions, or that some naively defined properties do not hold with our machine. In order to reflect these findings, we continuously modified the definition of the term-updating function and properties in Section 4.

The predicate *isCorrect()*, which is the conjunction of all the resulting properties, is strong enough to prove the pipeline correctness diagram, but weak enough to hold for any reachable pipeline states. This definition is a result of our interaction with the theorem prover. Also, as result of this interaction, we were forced to add lemmas for some sub goals to continue the prove process. The following is an example.

```
(lem {ps ; ps'}
  ((((isCorrect(ps)) =>
  ((NewSeqState(proj(flush(ps)))) =
  (proj(flush(NewPipelineState(ps)))))) &
  ((isCorrect(ps')) =>
  ((NewSeqState(proj(flush(ps')))) =
  (proj(flush(NewPipelineState(ps')))))))) =>
  ((isCorrect(ps ps')) =>
  ((NewSeqState(proj(flush(ps ps')))) =
  (proj(flush(NewPipelineState(ps)))))))
  to (1 . 3 . 1) .)
```

This is a lemma added to the sub-goal 1 . 3 . 1 (this is the ITP tree representation of sub-goals), which means that if Theorem 1 is verified for the pipeline design states *ps* and *ps'*, it will be verified for their union (the union is a constructor operator) *ps ps'*, in the condition that these states (*ps*, *ps'* and *ps ps'*) are all correct representations.

6. Conclusion

We have verified the behavioural equivalence of our pipelined machine with respect to its sequential specification using ITP theorem prover. Our machine design is certainly not the most complex pipelined machine that has been formally specified and verified. For the sake of the verification, we introduced the algebraic specification as a new level of machine abstraction. It allows us to directly reason about the instructions executed by the pipeline machine. With help of the algebraic specification, we defined various properties about the pipelined machine design. These properties are proved to hold for reachable pipelined states.

The proof are conducted incrementally by examining the related micro-architectural components. We adopted a correctness criteria that is defined using the idea of pipeline flushing. This allowed us to express the correctness of the pipelined machine containing complex control logic. The correctness criteria is verified using the pipeline properties of the reachable pipeline states. Since this approach does not appear to suffer the computational cost explosion of the other methods, we hope that our methodology will allow us to verify bigger and more complex examples with new features like out-of-order and speculative execution.

References

1. D. L. Beatty. A Methodology for formal Hardware Verification, with Application to Microprocessors. PhD thesis, School of Computer Science, Carnegie Mellon University, Aug. 1993.
2. R. E. Bryant, S. German, and M. N. Velev. Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions. Tableaux '99, June 1999.
3. J. R. Burch. Techniques for verifying superscalar microprocessors. In 33rd Design Automation Conference (DAC '96), Pages 552-557. June 1996.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In Computer-Aided Verification (CAV'94), volume 818 of LNCS, pages 68-80. Springer-Verlga, 1994.
5. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti- Oliet, J. Meseguer, and J. Quesada. Maude: Specification and Programming in Rewriting Logic, Computer Science Laboratory, SRI International, 1999.
6. M. Clavel. The ITP Tool, Department of Philosophy, University of Navarre, 2000.
7. A. J. Cohn. A proof correctness of the Viper microprocessors. The first level. In G. Birtwistle and P. A. Subrahmanyam, editors, VLSI Specification, Verification and Synthesis, pages 27-72. Kluwer, 1988.
8. A. J. Cohn Correctness properties of the Viper block model: The second level. In G. Birtwistle, editor, proceedings of the 1988 Design Verification Conference. Springer-Verlag, 1989.
9. R. B. Jones and D. L. Dill. Efficient Validity Checking for Processor Verification. In IEEE International Conference on Computer-Aided Design (ICCAD), 1995.
10. J. L.Hennessy, and D. A. Patterson, Computer Architecture: A Quantitative Approach, 2 nd edition Morgan- Kaufmann, San Francisco, 1996.
11. J. Sawada. Verification of a simple pipelined machine model. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, Computer-Aided Reasoning: ACL2 Case Studies, pages 137-150. Kluwer Academic Press, 2000.

Appendix

The following is some Maude code for our pipelined machine specification.

```
(goal (fmod PIPELINE is
. . .
op FetchStage : PipelineState
  ProgramCounter Bool -> PipelineState .
eq FetchStage(s, pc, stall) =
  if stall or Stall(s) then nil
  else <Inst: Fetch(updatePC(s,pc)); PC:
    updatePC(s,pc); Stg: IF>
  fi .
eq FetchStage(nil, pc, stall) =
  if stall then nil
  else <Inst: Fetch(pc); PC: pc; Stg: IF>
  fi .

op DecodeStage : PipelineState
  RegisterFile -> PipelineState .
eq DecodeStage(<Inst: I; PC: pc; Stg: IF>
  s, rf) =
  if Stall(<Inst: I; PC: pc; Stg: IF> s)
  then <Inst: I; PC: pc; Stg: IF>
  else <Inst:I; PC:pc; Stg:DC;
    A:readRegFile(I,'A,  updateRF(rf,s));
    B:readRegFile(I,'B,updateRF(rf, s));
    Imm: Immediate(I)>
  fi .
eq DecodeStage(s, rf) = nil .

op ExecuteStage : PipelineState ->
  PipelineState .
eq ExecuteStage(s <Inst: I; PC: pc;
  Stg: DC; A: a; B: b; Imm: imm> s') =
  if InstructionType(I) == LOAD then
    <Inst: I; PC: pc; Stg: EX; A: a; B: b;
    Imm: imm; AluOut: a + imm; Cond: false>
  else
    if InstructionType(I) == LOAD-IMM then
      <Inst: I; PC: pc; Stg: EX; A: a; B: b;
      Imm: imm; AluOut: imm; Cond: false>
    else if InstructionType(I) == STORE then
      <Inst: I; PC: pc; Stg: EX; A: a; B: b;
      Imm: imm; AluOut: a + imm ;
      Cond: false>
    else
      if InstructionType(I) == ALU-OP then
        <Inst: I; PC: pc; Stg: EX; A: a;
        B: b; Imm: imm; AluOut: a OP b;
        Cond: false>
      else
        if InstructionType(I) == ALU-IMM
        then
          <Inst: I; PC: pc; Stg: EX; A: a;
          B: b; Imm: imm; AluOut: a OP imm ;
          Cond: false>
        else --- Jump instruction
          <Inst:I; PC:pc; Stg:EX; A:a;
          B:b; Imm:imm; AluOut:
        updatePC(s', pc') + imm;
          Cond:a OP 0>
  fi fi fi fi fi fi .
eq ExecuteStage(s) = nil .

op MemoryStage : PipelineState ->
  PipelineState .
eq MemoryStage(s <Inst: I; PC: pc; Stg: EX;
  A: a; B:b; Imm:imm; AluOut: aluout;
  Cond:c> s') =
  if InstructionType(I) == LOAD or
      InstructionType(I) == LOAD-IMM then
    <Inst:I;  PC:pc;  Stg: MEM; A:a; B:b;
    Imm:imm; AluOut:aluout;
    Cond:c; LMD:Load(aluout,updateDM(s',
    dm)) >
  else <Inst: I; PC: pc; Stg: MEM; A: a;
    B: b; Imm: imm; AluOut: aluout;
    Cond: c; LMD: 0>
  fi .
eq MemoryStage(s) = nil .
op WriteBackStage : PipelineState ->
  PipelineState .
eq WriteBackStage(s <Inst:I; PC:pc;
  Stg:MEM; A:a; B:b; Imm:imm;
  AluOut:aluout; Cond:c;LMD:lmd> s') =
 <Inst: I; PC: pc; Stg: WB; A: a; B: b;
  Imm: imm; AluOut: aluout; Cond: c;
  LMD: lmd >

op updatePC : PipelineState
  ProgramCounter -> ProgramCounter .
eq updatePC(s <Inst:I; PC:pc; Stg:MEM; A:a;
  B:b; Imm:imm; AluOut:aluout; Cond:c;
  LMD:lmd> s', pc') =
  if c then aluout
  else pc' + 1
  fi .

op updateDmem : PipelineState DataMemory ->
  DataMemory .
eq updateDmem(s <Inst:I; PC:pc; Stg:MEM;
  A:a; B:b; Imm:imm; AluOut:aluout; Cond:c;
  LMD:lmd> s', dm) =
  if InstructionType(I) == STORE then
    write(dm, aluout, b)
  else dm
  fi .

op updateRegfile : PipelineState
  RegisterFile -> RegisterFile .
eq updateRegfile(s <Inst:I; PC:pc; Stg:WB;
  A:a; B:b; Imm:imm; AluOut:aluout; Cond:c;
  LMD:lmd>, rf) =
  if InstructionType(I) == ALU-OP or
    InstructionType(I) == ALU-IMM then
    write(rf, dest(I), aluout)
  else if InstructionType(I) == LOAD then
          write(rf, dest(I), lmd)
        else rf
        fi
  fi .

op Init : -> PipelineDesignState .
eq Init = <PipeState: nil; PC: 0; regfile:
  0 . 0; dmem: 0 . 0 . 0 . 0 . 0> .

op NewPipeState : PipelineDesignState
  Bool -> PipelineDesignState .
eq NewPipeState(<PipeState: nil; PC: pc;
  regfile: rf; dmem: dm>, stall) =
  <PipeState: FetchStage(nil, pc, stall)
  PC: updatePC(nil, pc);
  regfile: updateRegfile(nil, rf);
  Dmem: updateDmem(nil, dm)> .
eq NewPipeState(<PipeState: s; PC: pc;
  regfile: rf; dmem: dm>, stall) =
  <PipeState: (FetchStage(s, pc, stall)
  DecodeStage(s, rf)
  ExecuteStage(s)
  MemoryStage(s)
  WriteBackStage(s));
  PC: updatePC(s, pc);
  regfile: updateRegfile(s, rf);
 Dmem: updateDmem(s, dm)> .
. . .
endfm) |- {ps} ((isCorrect(ps)) =>
   ((NewSeqState(proj(Flush(ps)))) =
   (proj(Flush(NewPipelineState(ps))))))
.)
```