On Proving with Event-B that a Pipelined Processor Model Implements its ISA Specification

John Colley[1], Michael Butler[2]

[1] University of Southampton,
School of Electronics and Computer Science
Southampton, SO17 1BJ, UK
`jlc05r@ecs.soton.ac.uk`
[2] University of Southampton,
School of Electronics and Computer Science
Southampton, SO17 1BJ, UK
`mjb@ecs.soton.ac.uk`

Abstract. Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming. One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed. Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifies the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. Microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

Keywords. Event-B, Pipeline, ISA

1 Introduction

Microprocessor pipelining is a well-established technique that improves performance and reduces power consumption by overlapping instruction execution. Modern System-on-Chip microprocessors used for mobile applications have very stringent power consumption requirements and are typically based on the 5-stage DLX microprocessor [1]. From the Instruction Set Architecture (ISA) specification, a pipelined microarchitecture is developed that implements the specification. Verifying, however, that an implementation meets this ISA specification is complex and time-consuming. Current verification techniques are predominantly test based within a Register Transfer Level (RTL) simulation and synthesis flow.

One of the key verification issues that must be addressed is that of overlapping instruction execution. This can introduce hazards where, for instance, a new instruction reads the value from a register which will be written by an earlier instruction that has not yet completed. These are termed Read-After-Write (RAW) data hazards [1]. The presence of hazards depends on the instruction mix presented to the microprocessor and pseudo-random test generation techniques have been used in an attempt to achieve adequate test coverage of instruction combinations [2], [3] .

Formal techniques, using both model checking and theorem proving, have been used in microprocessor verification, but as an adjunct to the simulation-based flow. These techniques are applied after the design is completed in the hope of detecting errors not discovered by testing. Higher-level hardware description languages such as Bluespec [4] and CAL [5], which provide an automatic synthesis route to RTL, can speed up the design process, but it is the verification costs that dominate in the overall flow and the bulk of the verification must still be done at the Register Transfer Level.

Event-B [6], [7] is a proof-based modelling language and method that enables the development of specifications using refinement. The Rodin platform [8] is the Eclipse-based IDE that provides support for Event-B refinement and mathematical proof. Using Event-B's support for refinement with automated proof, a method is explored where the abstract machine represents directly an instruction from the ISA that specifies the effect that the instruction has on the microprocessor register file. Refinement is then used systematically to derive a concrete, pipelined execution of that instruction. At each refinement step the importance is shown of addressing the inherent simultaneity that characterises the pipelined behaviour and, in particular, the effects that feedback has in pipeline construction.

To illustrate the method, the *register/register arithmetic* instruction of a typical System-on-Chip (SoC) microprocessor is chosen that can exhibit RAW data hazards with overlapping execution. The technique, termed *forwarding*, where intermediate values are fed back to a stage that needs them, is employed in modern microprocessors to provide a very efficient means of managing RAW hazards [1]. Debugging the forwarding logic has, however, been found to be difficult and expensive [9] . With the introduction of appropriate invariants in our approach, it is shown that the concrete, pipelined refinement will not preserve these invariants unless the RAW hazards are detected and managed appropriately.

The concrete Event-B model implements forwarding in a way that corresponds directly to the techniques used in microprocessor design and is proved, automatically, in the Rodin environment to be a correct refinement of the abstract ISA specification. Thus, microarchitectural considerations are raised to the specification level and design choices can be verified much earlier in the flow. The concrete model also has a direct correspondence to an equivalent hardware description in the high-level languages Bluespec and CAL, which like Event-B are based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

2 An Overview of Event-B

In Event-B, an abstract model comprises a *machine* that specifies the high-level behaviour and a *context*, made up of sets, constants and their properties, that represents the type environment for the high-level machine. The machine is represented as a set of *state variables*, v and a set of events, *guarded atomic actions*, which modify the state. If more than one action is enabled, then one is chosen non-deterministically for *execution*, an observable transition on the state variables which must preserve an *invariant* on the variables, *I(v)*. A more concrete representation of the machine may then be created which refines the abstract machine, and the abstract context may be extended to support the types required by the refinement. *Gluing invariants* are used to verify that the concrete machine is a correct refinement of the abstract. Gluing invariants give rise to proof obligations for pairs of abstract and corresponding concrete events. Events may also have parameters which take, non-deterministically, the values that will make the guards in which they are referenced true.

An event can be represented by the *generalized substitution*,

$$\boxed{\textbf{any } x \textbf{ where } P(x,v) \textbf{ then } v := F(x,v) \textbf{ end}}$$

where x represents the event parameters and v represents the value of the machine state variables. Informally, this event can be fired provided that the guard *P(x, v)* can be satisfied for some value x. The details are explained in [10] .

3 Modelling the Arithmetic Instruction

3.1 The Abstract ISA Model

The structure of a *register/register arithmetic* instruction associates the opcode with a destination register *Rr* and two source registers *Ra* and *Rb*. The Event-B *context*, *PIPEC*, for the arithmetic instruction therefore defines a set of operations *Op*, the type *Register*, the subset of operations that are of type *register/register arithmetic*, *ArithRRop*, and the relationship between the fields of the arithmetic instruction and their associated registers. The conventions of [11] are followed to model operation fields. The context also defines *No Operation*, *NOP*.

```
CONTEXT  PIPEC
SETS
    Op
CONSTANTS
    Register
    Rr
    Ra
    Rb
    NOP
    ArithRROp
AXIOMS
    axm1 :  Register ⊆ ℕ
    axm2 :  Rr ∈ Op → Register
    axm3 :  Ra ∈ Op → Register
    axm4 :  Rb ∈ Op → Register
    axm5 :  ArithRROp ⊆ Op
    axm6 :  NOP ∈ Op
    axm7 :  NOP ∉ ArithRROp
END
```

The abstract machine, *PIPEM*, defines the register file *Regs* and a single event *ArithRR* that specifies the effect that execution of the instruction has on the register file. For simplicity, the addition operation is shown, but this can more generally be represented by an uninterpreted function [12] without affecting the proof approach used. The parameter *pop* specifies the environment for the event; given an instruction of type *ArithRROp*, the state of the register file will be updated according to that instruction.

```
MACHINE  PIPEM
SEES  PIPEC
VARIABLES
    Regs
INVARIANTS
    inv1 :  Regs ∈ Register → ℤ
EVENTS
Initialisation
    begin
        act1 :  Regs := Register × {0}
    end
Event  ArithRR ≙
    any
        pop
    where
        grd1 :  pop ∈ ArithRROp
    then
        act1 :  Regs(Rr(pop)) := Regs(Ra(pop)) + Regs(Rb(pop))
    end
END
```

The microarchitecture of the abstract machine is shown in Figure 1.

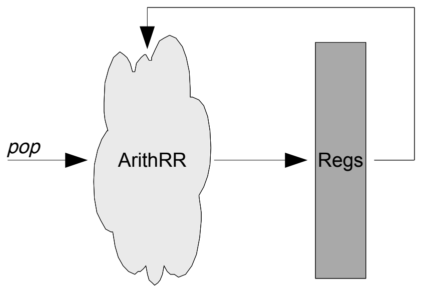

Fig. 1. Abstract Machine: Microarchitecture

3.2 The First Refinement: a 2-stage pipeline

A 2-stage pipeline is now introduced which refines the abstract machine. The second pipeline stage is a concrete representation of the *Write Back (WB)* stage while the first stage is still abstract, representing the *Fetch/Decode/Execute* operations of the pipeline.

MACHINE PIPER1
REFINES PIPEM
SEES PIPEC
VARIABLES
    `Regs`
    `EXop`
    `ALUout`
INVARIANTS
    `inv1` : $EXop \in Op$
    `inv2` : $ALUout \in \mathbb{Z}$
    `inv3` : $ALUout = Regs(Ra(EXop)) + Regs(Rb(EXop))$
EVENTS
Event $FDEXWB \mathrel{\widehat{=}}$
refines *ArithRR*
    any
        ppop
    where
        `grd1` : $EXop \in ArithRROp$
        `grd2` : $ppop \in ArithRROp$
        `grd3` : $Rr(EXop) \neq Ra(ppop)$

```
        grd4 : $Rr(EXop) \neq Rb(ppop)$
    with
        pop : pop = EXop
    then
        act1 : $Regs(Rr(EXop)) := ALUout$
        act2 : $ALUout := Regs(Ra(ppop)) + Regs(Rb(ppop))$
        act3 : $EXop := ppop$
    end
END
```

Two new variables, *ALUout* and *EXop* are introduced to represent the *EXWB* pipeline registers. The parameter *pop* of the abstract *ArithRR* event is bound to the concrete register *EXop* using an Event-B *witness* and a new parameter *ppop* represents the environment of the refined event, *FDEXWB*. The *FDEXWB* event models the simultaneous execution of both pipeline stages. The microarchitecture of the refined machine is shown in Figure 2.

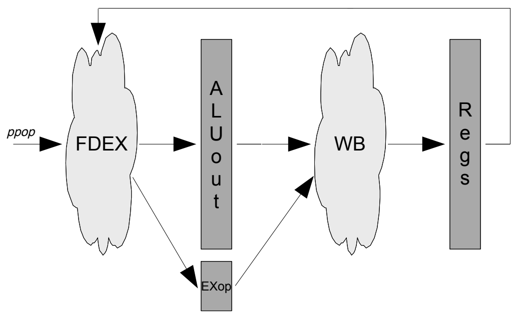

Fig. 2. Refined Machine: Microarchitecture

It is now necessary to introduce the *gluing invariant* to establish that this is a correct refinement of the abstract machine. To preserve the meaning of the abstract specification, the new variable *ALUout* must always have the value *Regs(Ra(EXop)) + Regs(Rb(EXop))*, as represented by the invariant *inv3*. The Rodin prover, however, shows that this invariant is not preserved by the refined machine. The abstract *FDEX* pipeline stage simultaneously reads the register file while the *WB* stage is writing to it. If the location being read is the same as that being written, a *Read After Write (RAW)* data hazard will be encountered and the wrong value will read by the first pipeline stage. This inherent feedback in the pipelined implementation must be addressed explicitly if it is to meet its specification.

3.3 Detecting the RAW Hazard

The abstract *FDEX* pipeline stage may only read from the source registers *Ra* and *Rb* if they do not coincide with the target register *Rr* of the previous instruction, represented by *Rr(EXop))*. Two new guards are introduced into the refined event to meet this requirement.

grd1 : ...
grd2 : ...
grd3 : $Rr(EXop) \neq Ra(ppop)$
grd4 : $Rr(EXop) \neq Rb(ppop)$

The Rodin prover now shows that the invariant $ALUout = Regs(Ra(EXop)) + Regs(Rb(EXop))$ is preserved by the refined machine.

3.4 Dealing Correctly with the RAW Hazard

It is now necessary to deal with the cases where a hazard is encountered on register *Ra* alone, on register *Rb* alone and on both registers *Ra* and *Rb*. In each case, the required value(s) can be read from the *ALUout* register. This corresponds directly to the *forwarding* technique used in microprocessor design. Three extra events are introduced to deal with each case. For instance, for the hazard on register *Ra*, the guards of the event are

grd3 : $Rr(EXop) = Ra(ppop)$
grd4 : $Rr(EXop) \neq Rb(ppop)$

and the associated action now reads the value of *Ra* from *ALUout*.

act2 : $ALUout := ALUout + Regs(Rb(ppop))$

The Rodin prover shows that, for each case, the invariant is preserved. The microarchitecture of the modified refined machine is shown in Figure 3.

3.5 Further Refinements

The refinement process can continue, systematically, until all the pipeline stages are represented in concrete form. At each step, the gluing invariants will ensure that the refinement implements its predecessor.

In the second refinement, the concrete Execute *(EX)* stage is introduced together with the *IDEX* pipeline registers. The registers *A* and *B* store the values of *Ra* and *Rb* respectively. Four events in the abstract *Fetch/Decode* stage are needed to deal with the possible data hazard combinations and two new gluing invariants,

inv1 : $A = Regs(Ra(IDop))$
inv2 : $B = Regs(Rb(IDop))$

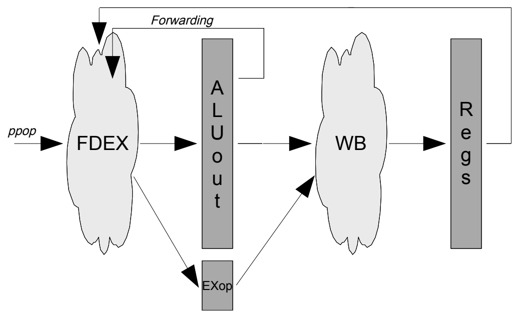

Fig. 3. Refined Machine with forwarding: Microarchitecture

ensure that the data hazards are dealt with correctly. When combined with the four *EXWB* events, this gives a total of sixteen events.

In the third refinement, the concrete Instruction Fetch *IF* and Instruction Decode *ID* are established.

To generalise this approach for uninterpreted arithmetic operations, the action

$\texttt{act1}: \ Regs(Rr(p)) := Regs(Ra(p)) + Regs(Rb(p))$

can be replaced with

$\texttt{act1}: \ Regs(Rr(p)) := fop(Regs(Ra(p)) \mapsto Regs(Rb(p)))$

where

$\texttt{grd1}: \ fop = func(p)$

and

$\texttt{axm8}: \ func \in Op \rightarrow Register$

is a field of the arithmetic instruction. The proofs with arbitrary arithmetic operations are still automatic.

The final, concrete pipeline is represented by sixteen events and all the proof obligations generated are discharged automatically by the Rodin tool, as shown in Table 1.

4 Related Work

Early work in the formal verification of microprocessors was focused on simple, non-pipelined processors described at the Register Transfer Level (RTL). In [13]

	Total no. of proof obligations	Discharged Automatically
Abstract Model	3	3
1st Refinement	33	33
2nd Refinement	192	192
3rd Refinement	115	115

Table 1. Pipeline Proofs

the RTL is represented in the ML programming language and the HOL proof assistant system [14] used to discharge the proofs.

In [12] and [15] the representation of the processor is raised to the Instruction Set Architecture (ISA) level and the techniques described focus on the formal verification of the control logic of first a 3-stage pipelined ALU and then the full 5-stage DLX processor. ALU operations are represented as uninterpreted functions. In order to show that the pipelined processor will behave in the same way as a notional non-pipelined version, the concept of pipeline *flushing* is introduced. *Stall* instructions are introduced at the pipeline input to ensure that each instruction is completed before the next is initiated. The notion of *refinement maps* are introduced in [16] and [17] to extend the flushing concepts of Burch and Dill to more complex 3 and 10-stage pipelines, using the ACL2 functional programming language and theorem prover [18].

[19] focuses its attention on the formalization of the pipeline hazards that can occur when multiple instructions are executed at once in the DLX pipeline. Structural, data and control hazards are represented and checked using the HOL verification system [14]. Incremental design techniques with refinement are described in [20] to show that a notional DLX pipeline that executes one instruction at a time can be refined to a pipeline that executes 5 instructions at each clock cycle and manages structural hazards does not encounter a sequence of instructions that would incur data or control hazards. This pipeline is then further refined to model the data and control hazards. Abstract State Machines (ASMs) are used to represent the DLX instructions. In [9], a tool that takes a sequential model of the DLX pipeline, which is assumed to be correct, and adds the forwarding logic is described. The tool also provides a proof of correctness for the generated hardware. Our approach is the only one that starts with an abstract ISA specification and proves, systematically, that the concrete, concurrent pipeline model derived from the ISA implements that specification.

5 Conclusions

A method has been explored, using the *register/register arithmetic* instruction as an example, to show that the ISA specification of the instruction can be refined systematically to a pipelined model that can be proved to implement its ISA specification. The method ensures, through the introduction of gluing invariants at each stage, that microarchitectural considerations are addressed early in the design flow. Different microarchitectures may be explored and verified at the specification level. Stepwise refinement allows us to manage the multiplicity of cases caused by pipeline data hazards. The models have been developed using the Rodin Platform and all the generated proof obligations are discharged automatically by the tool.

Current work is focused on managing the effect of branch instructions on correct pipeline execution. The techniques described have been used to prove that the pipeline program counter is updated correctly according to the branch instruction ISA specification. Gluing invariants are being developed to ensure that instructions that have been fetched speculatively are not executed when a branch is encountered.

A disadvantage of our approach is that we need to specify separate pipeline stages with a single event. We are exploring a technique that uses refinement and decomposition to create separate events for each stage once the gluing invariants have been proved.

In common with Bluespec and CAL, Event-B is based on guarded atomic actions. The method proposed therefore has the potential to be integrated into an existing high-level synthesis methodology, providing an automated design and verification flow from high-level specification to hardware.

References

1. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann (2006)
2. Hollander, Y., Morley, M., Noy, A.: The e language: A fresh separation of concerns. Proceedings of TOOLS **38** (2001)
3. Haque, F., Michelson, J.: Art of Verification with VERA. Verification Central (2001)
4. Nikhil, R.: Bluespec System Verilog: efficient, correct RTL from high level specifications. Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on (2004) 69–70
5. Bhattacharyya, S.S., Brebner, G., Janneck, J.W., Eker, J., von Platen, C., Mattavelli, M., Raulet, M.: Opendf: a dataflow toolset for reconfigurable hardware and multicore systems. SIGARCH Comput. Archit. News **36** (2008) 29–35
6. Abrial, J., Mussat, L.: Introducing dynamic constraints in B. B **98** (1998) 83–128
7. Hallerstede, S.: Justifications for the Event-B Modelling Notation. In: B 2007: Formal Specification and Development in B. (2007)
8. Abrial, J., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: International Conference on Formal Engineering Methods (ICFEM). (2006)
9. Kroening, D., Paul, W.: Automated pipeline design. In: Proceedings of the 38th conference on Design automation, ACM New York, NY, USA (2001) 810–815
10. Abrial, J.: Rigorous Open Development Environment for Complex Systems: event B language. (2005)
11. Evans, N., Butler, M.: A Proposal for Records in Event-B. FM (2006) 21–27
12. Burch, J., Dill, D.: Automatic verification of Pipelined Microprocessor Control. Proceedings of the 6th International Conference on Computer Aided Verification (1994) 68–80

13. Joyce, J., Birtwistle, G., Gordon, M.: Proving a Computer Correct in Higher Order Logic, Report No. 100. Computer Laboratory, Cambridge University (1986)
14. Gordon, M., Melham, T.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press New York, NY, USA (1993)
15. Jones, R., Dill, D., Burch, J.: Efficient validity checking for processor verification. In: IEEE International Conference on Computer-Aided Design, San Jose, California, USA (1995)
16. Manolios, P.: Correctness of pipelined machines. Formal Methods in Computer-Aided Design–FMCAD **1954** (2000) 161–178
17. Manolios, P., Srinivasan, S.: A computationally efficient method based on commitment refinement maps for verifying pipelined machines models. ACM-IEEE International Conference on Formal Methods and Models for Codesign (2005) 189–198
18. Kaufmann, M., Moore, J.: Industrial proofs with acl2. Technical report, University of Texas (2004)
19. Tahar, S., Kumar, R.: Formal Verification of Pipeline Conflicts in RISC Processors. Proc. European Design Automation Conference (EURO-DAC94), Grenoble, France, September (1994) 285–289
20. Borger, E., Mazzanti, S.: A Practical Method for Rigorously Controllable Hardware Design. ZUM'97, the Z Formal Specification Notation: 10th International Conference of Z Users, Reading, UK, April 3-4, 1997: Proceedings (1997)