

Verifying a Simple Pipelined Microprocessor Using Maude

N. A. Harman*

Department of Computer Science, University of Wales Swansea, Singleton Park,
Swansea SA2 8PP

Abstract. We consider the verification of a simple pipelined microprocessor in *Maude*, by implementing an equational theoretical model of systems. *Maude* is an equationally-based language, with an efficient term rewriting implementation, and effective meta-level tools. Microprocessors and other systems are modelled as *iterated maps* operating in time over some state-set, and are related by means of data and abstraction maps, and correctness is reduced to state exploration by the choice of an appropriate *initialisation function*, ensuring/enforcing consistency of the *initial state*.

1 Introduction

This paper considers the verification of a simple pipelined microprocessor in *Maude* [3], an equational, algebraic language with strong meta-language tools and an efficient term rewriting implementation. Hardware systems, and models of hardware correctness, are represented within a well-developed set of mathematical tools, developed by application to case studies, and based on an equational, algebraic model. In a related paper [13], we consider the process of verification in *Maude* in more detail.

Microprocessors, and related systems, are modelled as *iterated maps*

$$\begin{aligned} F &: T \times A \rightarrow A, \\ F(0, a) &= h(a), \\ F(t + 1, a) &= f(F(t, a)), \end{aligned}$$

where T is a clock, A is the state-set, f is a *next-state function* defining state evolution, and h is an (optional) *initialisation function*, ensuring/enforcing consistency of *initial state* a . Initialisation function h is an important component of the verification process, and careful construction is essential (Sect. 5.4 and [9, 7]) to reduce formal verification to state exploration. In this paper, we do not consider input and output: however, they are easily accommodated [15, 7].

Maude was chosen as the appropriate tool to implement our theoretical model because (a) it has the same mathematical basis; (b) it is fast (approximately

* This work is supported by UK EPSRC grants GR/N15955 and GR/M82202

700K rewrites per second on a 700MHz Pentium III, when applied to hardware examples); (c) its meta-level tools allow proof strategies to be constructed quickly and flexibly; and (d) it is easy to learn. However, other tools could also be used (initial experiments were undertaken with PVS [26]).

This paper forms part of a series on theoretical models of microprocessors. In [14, 15] mathematical models of *microprogrammed* examples are considered. In [9, 11], *correctness models* and the formal verification process are examined. In [8, 10] models of *superscalar* processors are examined by means of a substantial example. An extended account of some of this work can be found in [7]. To date, our principle interest has been *theoretical* models of systems, their correctness, and their verification: principally microprocessors, but also include programming languages and their compilers [28], including the Java Virtual Machine [29]. Work has progressed on a set of mathematical tools for modelling behaviour and correctness in a modular and software tool-independent way. However, in this paper and in [13] we consider the implementation of these mathematical tools within Maude.

The structure of this paper is as follows. In Sect. 1.1 we consider related work. In Sect. 2 we introduce the required theoretical fundamentals. In Sect. 3 we introduce the Maude system. In Sect. 4 we introduce the architecture (specification) of a simple microprocessor SPM, in Maude. In Sect. 5 we introduce a pipelined implementation ACP. In Sect. 6 we consider the correctness and verification of ACP with respect to SPM. Finally, in Sect. 7 we summarise our techniques and their applicability.

1.1 Related Work

The main distinction between this and related work is that our principle interest is building theoretical/mathematical models of system, rather than on addressing [industrial] examples within software tools, where an important theme has been efficient verification strategies.

Interesting work on pipelined microprocessor verification includes [25] on AAMP5, a non-trivial, industrial example, and its verification in PVS [26] (recent accounts are [6, 27]: see also [17]); [31] on UINTA, a processor of moderate complexity, and its verification in HOL [12]; and [2] on a part of DLX [16]. A refinement of the approach in [2], more applicable to out-of-order systems and long pipelines is [19, 20]. In addition, work has been undertaken on the complex timing models of superscalar processors [30, 1, 5]: [18] additionally considers exception processing in such an environment. The work in [21, 4] uses *Hawk*, a variant of the functional language *Haskell*.

Generally, the intuitive models seen are conceptually similar to our own [14, 15, 8], though significant differences exist in the approach to time. Commonly, in pipelined systems, state elements in the specification are viewed as distributed in time in the implementation. We regard specification states as [some function of] state elements in the implementation at a single point in time (see [9, 7]). In addition, time is explicitly present in our model. Although explicit time is removed from the verification process for microprocessors (Sect. 2.4), recall

that our principle interest has been in theoretical models of systems, of which microprocessors are only one example.

2 Theoretical Preliminaries

Computer systems are modelled in a universal algebra framework: see (among many) [32, 24]. We define functions equationally, primarily using definition by cases and primitive recursion. Time is modelled using a *clock algebra* and computer systems are modelled with [many-sorted] *state algebras*.

A many-sorted algebra consists of *carrier sets* and *functions* ranging over the carrier sets: $(A_1, A_2, \dots, A_l \mid f_1, f_2, \dots, f_m)$, with carrier sets A_1, A_2, \dots, A_l and functions f_i of the form $f_i : A_{s_1} \times A_{s_2} \times \dots \times A_{s_n} \rightarrow A_s$, where $1 \leq i \leq m$ and $1 \leq s, s_j \leq l$ for $1 \leq j \leq n$, and each f_i is defined by one or more equations. If $n = 0$ then f_i is called a *constant*. Many-sorted algebras are straightforwardly translated into Maude modules.

2.1 Clocks and Iterated Maps

Systems operate in time, starting at time zero in an *initial* state $h(a)$ where h is some *initialisation function*. Future system states are determined by a *next-state function* f , enumerated by a clock algebra $T = (T \mid 0, +1)$.

Definition 1. *Let T be a clock and let A be any non-empty set representing a state-space. An iterated map $F : T \times A \rightarrow A$ is a primitive recursive function defined by*

$$\begin{aligned} F(0, a) &= h(a), \\ F(t + 1, a) &= f(F(t, a)) \end{aligned}$$

where $h : A \rightarrow A$ and $f : A \rightarrow A$ are respectively the initialisation and next-state functions of the state function F .

Initialisation function h (which may be the identify function) limits the number of initial states, and (if carefully chosen) acts as an invariant during verification.

2.2 Data and Timing Abstraction Mappings

Data abstraction maps are surjective functions $\psi : B \rightarrow A$ between two state-spaces. Data abstraction maps are commonly projections between two composite state-spaces, for example, a map ψ from $B = B_1 \times B_2 \times \dots \times B_m$ to $A = B_{i_1} \times B_{i_2} \times \dots \times B_{i_n}$ defined as follows

$$\psi(b_1, b_2, \dots, b_m) = (b_{i_1}, b_{i_2}, \dots, b_{i_n})$$

where $1 \leq i_j \leq m$, $1 \leq j \leq n$ and $n \leq m$.

Two clocks are related using a temporal abstraction map, or *retiming*¹:

¹ Not to be confused with the retimings of [22]

Definition 2. A retiming λ is a surjective and monotonic map between two clocks such that $\lambda(0) = 0$. The set of all retimings from clock S to clock T is denoted by $Ret(S, T)$. The immersion $\bar{\lambda}$ of a retiming $\lambda \in Ret(S, T)$ is defined by

$$\bar{\lambda}(t) = \text{least } s \in S \text{ such that } \lambda(s) = t.$$

The set of all immersions of retimings in $Ret(S, T)$ is denoted by $Imm(S, T)$.

Monotonicity ensures there is never a discrepancy, after abstraction, in the temporal ordering of events because, for all $s, s' \in S$ if $s' \geq s$, then $\lambda(s') \geq \lambda(s)$ where λ is a retiming.

Given two clocks S and T related by retiming $\lambda \in Ret(S, T)$, and a clock cycle $s \in S$, we commonly wish to identify the clock cycle $s' \in S$ such that s' is the first cycle of S where $\lambda(s') = \lambda(s)$.

Definition 3. The function $start : Ret(S, T) \rightarrow [S \rightarrow S]$ is defined by

$$start(\lambda) = \bar{\lambda}\lambda.$$

Definition 4. A state-dependent retiming $\lambda : A \rightarrow Ret(S, T)$ is a map from states to retimings. The set of all state-dependent retimings from state-space A to retimings in $Ret(S, T)$ is denoted by $Ret(A, S, T)$.

For each state of an implementation there is an associated state-dependent *uniform* retiming, defined in terms of a *duration* function over the state-space of F .

Definition 5. Let $F : S \times A \rightarrow A$ be an iterated map and $dur : A \rightarrow S^+$ be a map from states to a positive number of clock cycles. The uniform retiming with respect to F and dur from a clock S to a slower clock T , is the state-dependent retiming $\lambda \in Ret(A, S, T)$ such that, for all $a \in A$ and $t \in T$

$$\begin{aligned} \bar{\lambda}(a)(0) &= 0, \\ \bar{\lambda}(a)(t+1) &= dur(F(\bar{\lambda}(a)(t), a)) + \bar{\lambda}(a)(t) \end{aligned}$$

where $\bar{\lambda} \in Imm(A, S, T)$ is the immersion of λ . The singleton set containing the uniform retiming with respect to F and dur is denoted by $URet_F^{dur}(A, S, T)$.

Suppose that F represents the implementation of some system over a clock S , and that T is the (slower) clock of the corresponding specification. Then specification clock cycle $t+1 \in T$ lasts $dur(x)$ cycles of clock S , where $x = F(\bar{\lambda}(a)(t), a)$ is the state of F on clock cycle $\bar{\lambda}(a)(t) \in S$. That is, the cycle of implementation clock S corresponding with the start of the previous specification clock cycle $t \in T$. Note that dur is a function only of state, and consequently the number of cycles corresponding with any state is independent of the numerical value of $t \in T$.

2.3 Implementation Correctness

Correctness is defined in terms of the relationship between two algebras, representing implementation and specification. The state sequences specified by the implementation are mapped onto those of the specification by a data abstraction map ψ and a temporal abstraction map λ .

Definition 6. A state function $G : S \times B \rightarrow B$ is a correct implementation of iterated map $F : T \times A \rightarrow A$ with respect to data abstraction map $\psi : B \rightarrow A$ and a state-dependent retiming $\lambda \in \text{Ret}(B, S, T)$ if, and only if, for all $b \in B$ and $s = \text{start}(\lambda(b))(s)$

$$F(\lambda(b)(s), \psi(b)) = \psi(G(s, b)),$$

or, alternatively, if the following diagram commutes for all $b \in B$ and $s = \text{start}(\lambda(b))(s)$

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \uparrow (\lambda, \psi) & & \uparrow \psi \\ S \times B & \xrightarrow{G} & B. \end{array}$$

Correctness must hold at all clock cycles corresponding with specification states, expressed by $s = \text{start}(\lambda(b))(s)$.

2.4 Time-Consistency and the One-Step Theorems

Iterated map state functions are *time-consistent* if all states that may arise at times $s \in \bar{\lambda}(B)$ are legal initial states.

Definition 7. An iterated map $F : S \times A \rightarrow A$ is time-consistent with respect to a state-dependent retiming $\lambda \in \text{Ret}(A, S, T)$ if, and only if, for all $a \in A$ and $t_1, t_2 \in T$

$$F(s_1 + s_2, a) = F(s_1, F(s_2, a))$$

where $s_2 = \bar{\lambda}(a)(t_2)$ and $s_1 = \bar{\lambda}(F(s_2, a))(t_1)$.

That is, $h \circ f^{s_1+s_2} = h \circ f^{s_2} \circ h \circ f^{s_1}$.

Initialisation function h must be carefully chosen to ensure time-consistency. In practice, h may be complex and difficult to construct. In Sect. 5.4 we describe a systematic method that is sometimes applicable.

The following two results are called one-step theorems. Theorem 1 states that if $\lambda \in \text{Ret}(B, S, T)$ is a uniform retiming then time-consistency with respect to λ is sufficiently verified by examining the implementation at times $t = 0, 1$. Theorem 2 states that retiming uniformity and implementation time-consistency are sufficient conditions to enable correctness to be verified by examining times $t = 0, 1$.

Theorem 1. *If $F : S \times A \rightarrow A$ is an iterated map with initialisation function $h : A \rightarrow A$ and if $\lambda \in URet_F^{dur}(A, S, T)$ is a uniform retiming then F is time-consistent with respect to λ if, and only if, for all $a \in A$*

$$F(0, a) = h(F(0, a)), \text{ and } F(\bar{\lambda}(a)(1), a) = h(F(\bar{\lambda}(a)(1), a)).$$

Proof. See [7, 9].

Theorem 2. *Let $F : T \times A \rightarrow A$ and $G : S \times B \rightarrow B$ be iterated maps. Let $\psi : B \rightarrow A$ be a data abstraction map and let $\lambda \in URet_G(B, S, T)$ be a uniform retiming. If F is non-initialised, and G is time-consistent with respect to λ then G is a correct implementation of F if, and only if*

$$F(0, \psi(b)) = \psi(G(0, b)), \text{ and } F(1, \psi(b)) = \psi(G(\bar{\lambda}(b)(1), b)).$$

Proof. See [7, 9].

3 Introduction to Maude

Maude is an equationally-based, algebraic language with a term rewriting implementation [3]. The following simple algebra, or module, representing a memory, illustrates the main features of interest to us².

```
fmod MEM is
  protecting MACHINE-WORD .
  sorts Mem .

  op _[_] : Mem MAR -> Word .          *** Memory read
  op _[_/_] : Mem Word MAR -> Mem .    *** Memory write

  var M : Mem .
  vars A B : MAR .
  var W : Word .

  eq M[W / A][A] = W .
  ceq M[W / A][B] = M[B] if A /= B .
endfm
```

The module `MEM` imports a module `MACHINE-WORD` defining (among other things) sorts `MAR` and `Word` representing memory addresses and memory words respectively. `MEM` introduces sort `Mem`, and operations `_[_]` and `_[_/_]`, representing memory reading and writing, defined by a pair of equations (one of which is conditional). The operations are defined in mixfix syntax, where arguments will replace the `_` characters in the operation name.

² Maude also includes a mechanism for defining type hierarchies, which we do not use here.

4 SPM Architecture

SPM is a simple microprocessor architecture, with five instructions (**add**, **load**, **store**, **branch** and **set**), separate program and data memories md and mp , a general purpose register set reg and a program counter pc . We use separate data and program memories to simplify the process of mapping to a pipelined implementation with data and instruction caches. However, this is not necessary. SPM was first used in the form described here in [10, 7]: a previous version appeared in [8]. In addition to the pipelined implementation ACP (Sect. 5), which has been verified manually [7] and using Maude (Sect. 6), a superscalar version ACS exists [10, 7]. A variant of ACS is currently being verified in Maude³.

The SPM architecture is parameterised by three constants $r, m, w \in N^+$ which determine the number of general-purpose registers, the memory address space and the word size respectively. There are 2^r general-purpose registers, 2^m memory addresses, and memory words and registers are w -bit words. Informally, the SPM instructions are as follows.

add ra rb rc. $reg[c] := reg[a] + reg[b]; pc := pc + 1.$
branch addr. If $reg[0] = 0$, then $pc := pc + addr$; otherwise $pc := pc = 1.$
load ra addr. $reg[a] := md[addr]; pc := pc + 1.$
store ra addr. $md[addr] := reg[a]; pc := pc + 1.$
set ra val. $reg[a] := val; pc := pc + 1.$

The basic format of SPM instructions is shown in Figure 1, from which it can be seen that $w \geq \max(3 + 3r, 3 + r + m)$. The state-spaces for op codes, register and memory addresses, machine words, registers and memory are as follows:

$$\begin{aligned} OP &= W_3 - \{101, 110, 111\}, & RI &= W_r, \\ MAR &= W_m, & Word &= W_w, \\ Reg &= [RI \rightarrow Word], & Mem &= [MAR \rightarrow Word]. \end{aligned}$$

We define SPM in Maude as follows. The [omitted] module **MACHINE-WORD** defines sorts for $OP, \dots, Word$, together with the operations used to extract bit-fields in instruction decoding. Memory is defined by module **MEM** (Sect. 3), and registers by module **REG** (omitted, but similar to **MEM**).

```
*** State of SPM, together with
*** tupling and projection functions
fmod SPM-STATE is
  protecting MACHINE-WORD .
  protecting MEM .
  protecting REG .

  sort SPMstate .
```

³ Using the modified correctness definition of [8, 10, 7] for superscalar processors



Fig. 1. Format of SPM Instructions.

```

*** Create state tuple
op (_,_,-,_) : Mem Mem Bits Reg -> SPMstate .
*** Projection functions
ops mp_ md_ : SPMstate -> Mem .
op pc_ : SPMstate -> Bits .
op reg_ : SPMstate -> Reg .

var S : SPMstate .
vars MP MD : Mem .
var PC : Bits .
var REG : Reg .

eq mp(MP,MD,PC,REG) = MP .
eq md(MP,MD,PC,REG) = MD .
eq pc(MP,MD,PC,REG) = PC .
eq reg(MP,MD,PC,REG) = REG .
endfm

*** SPM - the Programmer's Level representation
fmod SPM is
  protecting SPM-STATE .

  *** State function
  op spm : MachineInt SPMstate -> SPMstate .
  *** Next-state function
  op next-spm : SPMstate -> SPMstate .

```

```

var SPM : SPMstate .
var T : MachineInt .
vars MP MD : Mem .
var PC : Bits .
var REG : Reg .

eq spm(0,SPM) = SPM .
ceq spm(T,SPM) = next-spm(spm(T - 1,SPM)) if T > 0 .

*** Addition
ceq next-spm(MP,MD,PC,REG) = (MP, MD, PC + 1,
  REG[REG[ra(MP[PC])] ++ REG[rb(MP[PC])] / rc(MP[PC])])
  if op(MP[PC]) == 0 .
*** [Taken] branch
ceq next-spm(MP,MD,PC,REG) = (MP, MD, PC + addr(MP[PC]), REG)
  if op(MP[PC]) == 1 and REG[0] == 0 .
*** [Not-taken] branch
ceq next-spm(MP,MD,PC,REG) = (MP, MD, PC + 1, REG)
  if op(MP[PC]) == 1 and REG[0] /= 0 .
*** Load
ceq next-spm(MP,MD,PC,REG) = (MP, MD, PC + 1,
  REG[MD[addr(MP[PC])]/ra(MP[PC])]) if op(MP[PC]) == (1 0) .
*** Store
ceq next-spm(MP,MD,PC,REG) =
(MP, MD[REG[ra(MP[PC])]/addr(MP[PC])], PC + 1, REG)
  if op(MP[PC]) == (1 1) .
*** Set
ceq next-spm(MP,MD,PC,REG) = (MP, MD, PC + 1,
  REG[val(MP[PC])/ra(MP[PC])]) if op(MP[PC]) == (1 0 0) .
endfm

```

5 The Pipelined Implementation ACP

The implementation ACP of SPM has a four-stage pipeline (see Figure 2) with the following stages.

Fetch. A single instruction is fetched from the instruction cache⁴ either using a *fetch program counter* fpc , or a branch target address generated by the execute unit. In normal operation, when the pipeline is full (i.e. a branch has not been taken in the past three cycles), $fpc = pc + 3$, where pc is the architectural program counter. In the event of a read-write conflict (Sect. 5.2), no instruction is fetched. The instruction is stored in an instruction register.

⁴ We omit the possibility of cache misses to simplify the example.

Decode. The contents of the instruction register in the Fetch Unit is decoded into five [overlapping] fields: *op*, representing the op-code; *ra*, *rb* and *rc* representing three register indices; and *val/addr* representing a memory address or immediate value. See Figure 1.

Execute. The instruction stored in the Decode Unit is executed, and the results encoded as a triple representing the *result*; the *destination* register/memory address; and the *unit* (if the result is to be stored in memory, register, or program counter; or if outcome is a failed conditional branch, or the pipeline stalls). Some elements of the execution triple are redundant in some circumstances.

Committal. Results from the Execution Unit are written to program counter, registers and/or data cache.

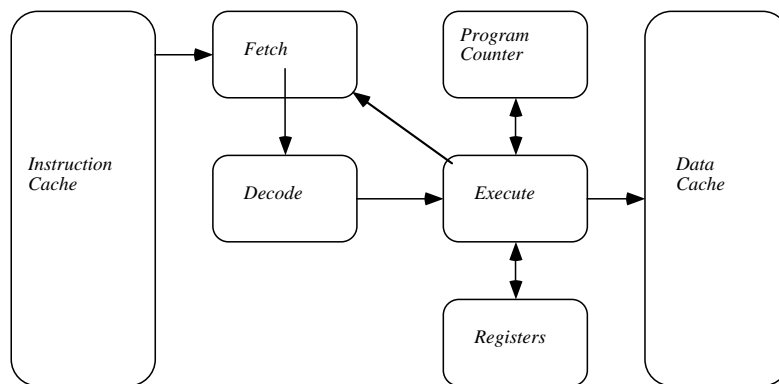


Fig. 2. Structure of ACP.

5.1 Pipeline States

The pipeline of ACP has four distinct states, identified by a *reset* counter and the *unit* field of the execution triple (and the relative values of *pc* and *fpc*).

Boot. In this state, which can only occur at start up, *reset* = 2, *fpc* = *pc*, *unit* is set to **wait** (i.e. no results to be committed), and the instruction register in the Fetch Unit and Decode Unit contain junk.

After Branch. This state is indistinguishable from one cycle after boot. In this state *reset* = 1, *fpc* = *pc* + 1, *unit* is set to **wait**, the instruction register in the Fetch Unit contains *mp[pc]*, and the Decode Unit contains junk.

Stall. This state is indistinguishable from two cycles after boot. In this state, *reset* = 0, *fpc* = *pc* + 2, *unit* is set to **wait**, the instruction register in the Fetch Unit contains *mp[pc + 1]*, and the Decode Unit the decoded form of *mp[pc]*.

Pipeline full. In this state, *reset* = 0, *fpc* = *pc* + 3, *unit* is set to something other than **wait** (indicating some result to be committed), the instruction register in the Fetch Unit contains *mp[pc + 2]*, and the Decode Unit the decoded form of *mp[pc + 1]*.

5.2 Pipeline Conflicts

There are a number of circumstances in which pairs of consecutive instructions can conflict. If the first instruction is a branch, then there is a *procedural dependency* between the instructions, and the second instruction will need to be discarded if the branch is taken by flushing the pipeline, and switching to the *after branch* state. There may also be *data dependencies* (or *RAW hazards*) between instruction pairs, if the second instruction requires the result of the first. In this case, the pipeline is suspended for one cycle by switching to the *stall* state, to allow the result of the first instruction to be committed before the second executes. The various data dependencies are illustrated in table 5.2 (‘-’ indicates no dependency). Note that it is possible to eliminate stalling the pipeline by

1st Inst.	2nd Instruction											
	add a2 b2 c2	branch a2	load a2 b2	store a2 b2	set a2 b2							
add a1 b1 c1	a2 ≠ c1	c1 ≠ 0	-	a2 ≠ c1	-							
	b2 ≠ c1											
load a1 b1	a2 ≠ a1	a1 ≠ 0	-	a2 ≠ a1	-							
	b2 ≠ a1											
store a1 b1	-	-	b2 ≠ b1	-	-							
set a1 b1	a2 ≠ a1	a1 ≠ 0	-	a2 ≠ a1	-							
	b2 ≠ a1											

Table 1. Possible data dependencies between instructions: note that data dependencies cannot occur when the first instruction is a branch.

providing mechanisms to permit internal *forwarding* of results before they are committed to registers/memory. However, we do not include such mechanisms here.

5.3 Formal Description of ACP

Space precludes including the full formal description of ACP in Maude (about 550 lines of with comments and whitespace). Instead, we give a partial definition in the notation used in the underlying mathematical model. This description of ACP first appeared in [7].

The state of ACP is the cartesian product of the states of each of its components:

$$State_{ACP} = Icache \times Ftch \times Dec \times Exec \times Reg \times MAR \times Dcache,$$

where

$$\begin{aligned} Icache &= Dcache = Mem, \\ Ftch &= Word \times MAR, \\ Dec &= OP \times RI \times RI \times RI \times MAR, \end{aligned}$$

$$\begin{aligned}
Exec &= Word \times MAR \times Unit \times Ctr, \\
Unit &= \{\mathbf{reg}, \mathbf{pc}, \mathbf{incpc}, \mathbf{dcache}, \mathbf{wait}\}, \\
Ctr &= \{0, 1, 2\}.
\end{aligned}$$

The iterated map $ACP : S \times State_{ACP} \rightarrow State_{ACP}$ is defined by

$$\begin{aligned}
ACP(0, \vec{\sigma}) &= \mathit{init}_{ACP}(\vec{\sigma}), \\
ACP(s+1, \vec{\sigma}) &= \mathit{next}_{ACP}(ACP(s, \vec{\sigma})),
\end{aligned}$$

where $\vec{\sigma} = (icache, \vec{f}, \vec{d}, \vec{e}, reg, pc, dcache)$, and

$\vec{f} = (ir, fpc)$, representing the instruction register and fetch program counter;
 $\vec{d} = (op, ra, rb, rc, addr)$, representing the decoded instruction fields; and
 $\vec{e} = (result, dest, unit, reset)$, representing the execution triple and
the pipeline state counter.

It remains to define init_{ACP} and next_{ACP} . The definition of init_{ACP} is central to the verification process, and is discussed in Sect. 5.4. Next-state function $\mathit{next}_{ACP} : State_{ACP} \rightarrow State_{ACP}$ is defined as follows:

$$\begin{aligned}
\mathit{next}_{ACP}(\vec{\sigma}) &= (icache, \mathit{Fetch}(\vec{\sigma}), \mathit{Decode}(\vec{\sigma}), \mathit{Execute}(\vec{\sigma}), \\
&\quad \mathit{Register}(\vec{\sigma}), \mathit{Counter}(\vec{\sigma}), \mathit{DCache}(\vec{\sigma})).
\end{aligned}$$

With the exception of $icache$, which does not change state, each unit of ACP has its own next-state function. Here, we only define the next-state function for the Execute unit.

The next-state function $\mathit{Execute} : State_{ACP} \rightarrow Exec$ generates an execution triple for each instruction, specifying result, destination and unit, together with the value of the $reset$ counter, controlling the state of the pipeline.

$$\mathit{Execute}(\vec{\sigma}) = \begin{cases} \mathit{exec}(\vec{d}, reg, pc, dcache), & \text{if not } \mathit{Conflict}(\vec{d}, \vec{e}) \text{ and } reset = 0; \\ (result, dest, \mathbf{wait}, 0), & \text{if } \mathit{Conflict}(\vec{d}, \vec{e}) \text{ and } reset = 0; \\ (result, dest, \mathbf{wait}, reset - 1), & \text{otherwise.} \end{cases}$$

In the event that the pipeline is not full ($reset > 0$), $\mathit{Execute}$ decrements $reset$ and prevents any results from being committed by setting $unit = \mathbf{wait}$. Otherwise, if the pipeline is full ($reset = 0$), then $\mathit{Execute}$ checks for conflicts and either executes an instruction (using exec), or stalls the pipeline. Subfunction

$Conflict : Dec \times Exec \rightarrow \mathbf{B}$ identifies conflicts between instructions, and is defined by

$$Conflict(\vec{d}, \vec{e}) = \begin{cases} tt, & \text{if } (unit = \mathbf{reg} \text{ and } ((op = \mathbf{branch} \text{ and } dest = 0) \text{ or} \\ & (op = \mathbf{add} \text{ and } (trim_{m \rightarrow r}(dest) = ra \text{ or } (trim_{m \rightarrow r}(dest) = rb)) \text{ or} \\ & (op = \mathbf{store} \text{ and } (trim_{m \rightarrow r}(dest) = ra \text{ or} \\ & \quad unit = \mathbf{dcache} \text{ and } (op = \mathbf{load} \text{ and } dest = addr)); \\ ff, & \text{otherwise.} \end{cases}$$

and $trim_{m \rightarrow r}$ truncates a binary word of length m to a word of length r . Subfunction $exec : Dec \times Reg \times MAR \times Dcache \rightarrow Exec$ executes the current instruction, and is defined as follows.

$$exec(\vec{d}, reg, pc, dcache) = \begin{cases} (reg[ra] + reg[rb], pad_{r \rightarrow m}(rc), \mathbf{reg}, 0), & \text{if } op = \mathbf{add}; \\ (result, pc + addr, \mathbf{pc}, 2), & \text{if } op = \mathbf{branch} \text{ and } reg[0] = 0; \\ (result, dest, \mathbf{incpc}, 0), & \text{if } op = \mathbf{branch} \text{ and } reg[0] \neq 0; \\ (dcache[addr], pad_{r \rightarrow m}(ra), \mathbf{reg}, 0), & \text{if } op = \mathbf{load}; \\ (reg[ra], addr, \mathbf{dcache}, 0), & \text{if } op = \mathbf{store}; \\ (pad_{m \rightarrow w}(addr), pad_{r \rightarrow m}(ra), \mathbf{reg}, 0), & \text{if } op = \mathbf{set}; \end{cases}$$

The functions $pad_{a \rightarrow b}$ extend binary words of length a to words of length b by adding leading zeros.

5.4 Constructing the Initialisation Function

While it is straightforward to define an initialisation function that is ‘correct’, care must be taken if the one-step theorems are to be applied, as the initialisation function $init_{ACP}$ must have the property $init_{ACP}(\vec{\sigma}) = \vec{\sigma}$, for any $\vec{\sigma} \in ACP(S \times State_{ACP})$. Given the complexity of the state of pipelined (and superscalar) systems, it is not easy to construct such initialisation functions: they must be able to identify and leave unchanged *all* legitimate states of a system. This problem is analogous to the construction of invariants and visible state predicates.

In the case of ACP, it is possible to systematically construct the initialisation function $init_{ACP}$ using the next-state function $next_{ACP}$. This is because the state of the pipeline of ACP is uniquely determined by the instructions currently in the pipeline. This is not always the case. Consider a superscalar processor, with multiple integer function units, where instructions are queued for execution. If instructions are dispatched to function units on the basis of queue lengths, to balance execution loads across units, then the state of the pipeline will depend on the number and distribution of instructions that have now left the pipeline. However, in ACP this is not the case and we can systematically define $init_{ACP}$.

The method used is as follows. Given the current state

$$\vec{\sigma} = (icache, \vec{f}, \vec{d}, result, dest, unit, reset, reg, pc, dcache),$$

we apply a reset function $BootState : State_{ACP} \rightarrow State_{ACP}$

$$BootState(icache, \vec{f}, \vec{d}, result, dest, unit, reset, reg, pc, dcache) = (icache, \vec{f}, \vec{d}, result, dest, \mathbf{wait}, 2, reg, pc, dcache)$$

that returns ACP to a boot state $\vec{\sigma}_0$ by setting $unit_0 = \mathbf{wait}$ and $reset_0 = 2$. We can check if $\vec{\sigma}$ is a legal *boot* state, using $Boot : State_{ACP} \rightarrow \mathbf{B}$

$$Boot(\vec{\sigma}) = (unit = BootState(\vec{\sigma})_{unit} \text{ and } reset = BootState(\vec{\sigma})_{reset}),$$

where $BootState(\vec{\sigma})_x$ projects out element x of $BootState(\vec{\sigma})$.

If ACP is not in a boot state, we can apply $next_{ACP}$ to $BootState(\vec{\sigma})$ and check if $\vec{\sigma}$ is an *after branch* state of ACP with $AfterBranch : State_{ACP} \rightarrow \mathbf{B}$

$$AfterBranch(\vec{\sigma}) = (\vec{f} = next_{ACP}(BootState(\vec{\sigma}))_{\vec{f}} \text{ and } unit = next_{ACP}(BootState(\vec{\sigma}))_{unit} \text{ and } reset = next_{ACP}(BootState(\vec{\sigma}))_{reset}).$$

Similarly, we can determine if ACP is in a *stall* state, or if the pipeline is *full* with $AfterConflict : State_{ACP} \rightarrow \mathbf{B}$ and $PipeFull : State_{ACP} \rightarrow \mathbf{B}$:

$$AfterConflict(\vec{\sigma}) = (\vec{f} = next_{ACP}^2(BootState(\vec{\sigma}))_{\vec{f}} \text{ and } \vec{d} = next_{ACP}^2(BootState(\vec{\sigma}))_{\vec{d}} \text{ and } unit = next_{ACP}^2(BootState(\vec{\sigma}))_{unit} \text{ and } reset = next_{ACP}^2(BootState(\vec{\sigma}))_{reset}),$$

$$PipeFull(\vec{\sigma}) = (\vec{\sigma} = next_{ACP}^3(BootState(\vec{\sigma}))).$$

We can now define an initialisation function that leaves legal initial states unchanged, and returns others to a boot state:

$$init_{ACP} : State_{ACP} \rightarrow State_{ACP},$$

$$init_{ACP}(\vec{\sigma}) = \begin{cases} \vec{\sigma}, & \text{if } AfterBranch(\vec{\sigma}) \text{ or } AfterConflict(\vec{\sigma}) \\ & \text{or } PipeFull(\vec{\sigma}); \\ BootState(\vec{\sigma}), & \text{otherwise.} \end{cases}$$

6 The Correctness Statement and Verification Process

To express the correctness of ACP with respect to SPM, we must construct data abstraction map ψ and retiming λ . The data abstraction map $\psi : State_{ACP} \rightarrow State_{SPM}$ is defined by

$$\psi(\vec{\sigma}) = (icache, dcache, pc, reg).$$

Note that this is an untypically simple data abstraction function: normally, we would expect some transformation of implementation state elements to be necessary. The retiming $\lambda \in URet_{ACP}^{dur}(State_{ACP}, S, T)$ is defined in terms of the following duration function $dur : State_{ACP} \rightarrow S^+$ as described in definition 5.

$$dur(\vec{\sigma}) = \begin{cases} 3, & \text{if } AfterBranch(\vec{\sigma}); \\ 2, & \text{if } AfterConflict(\vec{\sigma}); \\ 1, & \text{if } PipeFull(\vec{\sigma}); \\ 4, & \text{otherwise.} \end{cases}$$

To verify the correctness of ACP with respect to SPM using the one-step theorems 1 and 2, we need to prove the following.

$$next_{ACP}(0, \vec{\sigma}) = init_{ACP}(next_{ACP}(0, \vec{\sigma})) \quad (1)$$

$$next_{ACP}(dur(\vec{\sigma}), \vec{\sigma}) = init_{ACP}(next_{ACP}(dur(\vec{\sigma}), \vec{\sigma})), \quad (2)$$

$$next_{SPM}(0, \psi(\vec{\sigma})) = \psi(next_{ACP}(0, \vec{\sigma})) \quad (3)$$

$$next_{SPM}(1, \psi(\vec{\sigma})) = \psi(next_{ACP}(dur(\vec{\sigma}), \vec{\sigma})), \quad (4)$$

where $next_{SPM}$ was defined in Maude as `next-spm` in Sect. 4. Equations 1 and 2 discharge the obligations of theorem 1, and equations 3 and 4 discharge the obligations of theorem 2

6.1 Verification in Maude

Maude is not by itself a theorem proving system, and cannot automatically verify systems without direction. The simplest approach (and the most efficient in terms of number of rewrites) is to manually explore the cases, by defining modules containing sets of constants representing the values of, and relationships between, state elements for each case that needs to be considered. This was the first approach taken with this example. There are 53 cases to consider: three concern correctness and time-consistency when the pipeline is not full; six concern correctness and time-consistency when the pipeline is full at time $t = 0$ (two for branch, one each for the other instructions); and the remaining 44 concern correctness and time-consistency when the pipeline is full at time $t = 1$, and explore all possible cases of conflicting and non-conflicting instruction pairs. The verification process requires approximately 300K rewrites, taking less than a second on a 700MHz Pentium III with 1Gbyte of memory. A previous manual verification had been undertaken in [7]. Verification in Maude uncovered an off-by-one bug in the branch target address calculation, and an error in the initialisation function⁵. However, the process of manually defining cases is time-consuming, and potentially error-prone. The example here, with 53 cases, is about the largest that can be sensibly attempted in this way: the superscalar example from [10, 7] will require about 2000 cases.

⁵ This was a technical problem in the verification, and would not have affected any implementation.

To address this, we can build proof strategies within the Maude meta-level, that enables Maude modules to be treated as data types, and the rewriting process to be controlled. We can use these properties to construct a range of verification strategies, tailored to classes of example. In this case, we employed a simple, naive, strategy, and used Maude to automatically construct and check the tree representing all the subcases we are interested in, by defining operations to dynamically extend Maude modules by adding equations asserting each case, and directing the rewriting process. A more detailed description of this process can be found in [13]. The example was successfully re-verified using this method, requiring approximately 2.5M rewrites and taking about 4 seconds on the same 700MHz PIII.

7 Further Work and Considerations

The Maude strategy used for this example is experimental, and not particularly efficient. The same example was previously run successfully without running out of physical memory on a 64Mbyte machine, suggesting that there is scope to undertake significantly larger examples on existing hardware without addressing the efficiency of the proof strategy. However, a number of approaches are being followed to increase the size of examples that can be addressed.

- The existing proof strategy can be made more efficient. For example, by reducing the number of times identical sub-terms are evaluated.
- Different proof strategies can be considered. For example, to automatically generate (possibly in stages) the sets of case-defining constants used in the manually-directed proof.
- Combining the manual and automatic approaches, by defining sets of constants to identify *groups* of sub-cases, and automatically verifying all sub-cases within a group.
- To adopt techniques seen in the literature to reduce verification complexity (for example, [5]). Note that because our principle aim is building coherent *theoretical models* of systems, and their correctness and verification, rather than performing verifications *per se*, it will be necessary for us to integrate such efficiency-increasing techniques into our theoretical model first.

One of the key strengths of Maude is the ease with which proof strategies can be constructed: together with the efficiency of the underlying term-rewriting engine, this means that effective, specialised verification tools can be built relatively quickly. A memory system example currently being addressed will require induction: an inductive proof strategy is being currently being developed (based on a pre-existing example [23]). Other future work includes completing verification of ACS, the superscalar implementation of SPM, and addressing the high-level language and compiler examples of [28, 29]. In addition, theoretical work is underway on a model of operating system kernels: starting with a simple system involving multiple communicating processes. The ultimate intention is to build a unified theoretical model of systems from high-level languages to hardware, backed by an implementation in Maude.

References

1. J Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference*, 1996.
2. J Burch and D Dill. Automatic verification of pipelined microprocessor control. In D Dill, editor, *Proceedings of the 6th International Conference, CAV'94: Computer-Aided Verification*, pages 68 – 80. Lecture Notes in Computer Science 818, Springer-Verlag, 1994.
3. M Clavel, F Durán, S Eker, P Lincoln, N MartíOliet, J Meseguer, and J Quesada. Maude: Specification and programming in rewriting logic. Technical report, Computer Science Laboratory, SRI International, 1999.
4. B Cook, J Launchbury, and J Matthews. Specifying superscalar microprocessors in hawk. In M Sheeran, editor, *Workshop on Formal Methods for Hardware (Marstrand, Sweden)*, 1998.
5. D Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In A Camilleri M Srivas, editor, *Formal Methods in Computer-Aided Design*, pages 172 – 186. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
6. D Cyrluk, J Rushby, and M Srivas. Systematic formal verification of interpreters. In *IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 140 – 149, 1997.
7. A C J Fox. *Algebraic Representation of Advanced Microprocessors*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1998.
8. A C J Fox and N A Harman. An algebraic model of correctness for superscalar microprocessors. In *Formal Methods in Computer-Aided Design*, pages 346 – 361. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
9. A C J Fox and N A Harman. Algebraic models of correctness for microprocessors. Technical Report CSR 8-98 (accepted for *Formal Aspects of Computer Science*), University of Wales Swansea, 1998.
10. A C J Fox and N A Harman. Algebraic models of superscalar microprocessor implementations: A case study. In B Möller and J V Tucker, editors, *Prospects for Hardware Foundations*, pages 138 – 183. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
11. A C J Fox and N A Harman. Algebraic models of temporal abstraction for initialised iterated state systems: An abstract pipelined case study. Technical Report CSR 21-98 (submitted to *Acta Informatica*), University of Wales Swansea, 1998.
12. M J C Gordon and T F Melham. *Introduction to HOL*. Cambridge University Press, 1993.
13. N A Harman. Correctness and verification of hardware systems using maude. Technical Report Computer Science Report, University of Wales Swansea, 2000.
14. N A Harman and J V Tucker. Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica*, 33:421 – 456, 1996.
15. N A Harman and J V Tucker. Algebraic models of microprocessors: the verification of a simple computer. In V Stavridou, editor, *Proceedings of the 1995 IMA Conference on Mathematics for Dependable Systems*. Oxford University Press, 1997.
16. J L Hennessy and D A Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufman, 1996.
17. R Hosabettu, M Srivas, and G Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 122 – 134. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.

18. J Sawada W A Hunt. Processor verification with precise exceptions and speculative execution. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 135 – 147. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.
19. R B Jones, J U Skakkebæk, and D Dill. Reducing manual abstraction in formal verification of out-of-order execution. In G Gopalakrishnan and P Windley, editors, *Formal Methods in Computer-Aided Design, FMCAD 98*, pages 2 – 17. Springer-Verlag, Lecture Notes in Computer Science 1522, 1998.
20. J U Skakkebæk, R B Jones, and D Dill. Formal verification of out-of-order execution using incremental flushing. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 98 – 109. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.
21. S Krstic, B Cook, J Launchbury, and J Matthews. A correctness proof of a speculative, superscalar, out-of-order, renaming microarchitecture. Technical report, Oregon Graduate Institute, 1998.
22. C E Leiserson, F M Rose, and J B Saxe. Optimizing synchronous circuitry by retiming. In R Bryant, editor, *Third Caltech Conference on VLSI*, volume 1983, pages 87–116. Computer Science Press, 1803 Research Boulevard, Rockville MD 20850, 1983.
23. F Dur´ an M Clavel, S Eker, and J Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98*, 1998.
24. K Meinke and J V Tucker. Universal algebra. In T S E Maibaum S Abramsky, D Gabbay, editor, *Handbook of Logic in Computer Science*, pages 189 – 411. Oxford University Press, 1992.
25. S Miller and M Srivas. Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. In *Proceedings of WIFT 95, Boca Raton*, 1995.
26. S Owre, J Rushby, N Shankar, and M Srivas. A tutorial on using PVS. In *Proceedings of TPCD 94*, pages 258–279. Lecture Notes in Computer Science 901, Springer-Verlag, 1994.
27. M Srivas, H Rueß, and D Cyrluk. Hardware verification using PVS. In T Kropf, editor, *Formal Hardware Verification*, pages 156 – 205. Springer-Verlag, Lecture Notes in Computer Science 1287, 1997.
28. K Stephenson. *An Algebraic Approach to Syntax, Semantics and Compilation*. PhD thesis, University of Wales Swansea Computer Science Department, 1996.
29. K Stephenson. Algebraic specification of the Java virtual machine. In B Möller and J V Tucker, editors, *Prospects for Hardware Foundations*. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
30. P Windley and J Burch. Mechanically checking a lemma used in an automatic verification tool. In A Camilleri M Srivas, editor, *Formal Methods in Computer-Aided Design*, pages 362 – 376. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
31. P Windley and M Coe. A correctness model for pipelined microprocessors. In *Proceedings of the 2nd Conference on Theorem Provers in Circuit Design*, 1994.
32. M Wirsing. Algebraic specification. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 675 – 788. Elsevier, 1990.