An Algebraic Framework for Modelling and Verifying Microprocessors using HOL

Anthony Fox
Computer Laboratory, University of Cambridge

March 23, 2001

Abstract

This report describes an algebraic approach to the specification and verification of microprocessor designs. Key results are expressed and verified using the HOL proof tool. Particular attention is paid to the models of time and temporal abstraction, culminating in a number of *one-step* theorems. This work is then explained with a small but complete case study, which verifies the correctness of a datapath with microprogram control.

Contents

1 Introduction

This report describes a number of definitions and theorems, using the HOL proof tool, which constitute a structured framework for the specification and verification of microprocessor designs. A constructive approach is taken, both to the modelling of state systems, and to the modelling of data and temporal abstraction. For example, initialisation and next-state functions are used instead of invariants or predicates over valid state streams, and temporal abstraction is defined by means of duration functions.

An initialisation function $init : A \to A$ is used to map *arbitrary* states from some state-space A to *valid* initial states. For example, the pipeline of a pipelined processor cannot be filled arbitrarily because each pipeline stage (such as, fetch, decode and execute) is expected to be working on consecutive instructions. A trivial initialisation function for such a pipelined design would simply ensure that the pipeline was empty (flushed), ready to be filled with instructions. A more complex initialisation function may, instead of always resetting the pipeline, preserve states for which the pipeline is correctly filled, thus ensuring $init\ a = a$ wherever possible.

A next-state function $next : A \to A$ is used to specify the behaviour of a system over time. If the current state of the system is $a \in A$ then this will be followed by the state $next\ a$. For example, the next-state function in the specification of a computer architecture would model the state change caused by instruction execution.

This approach contrasts with less constructive (and more declarative) treatments, in which hardware is not modelled exclusively in terms of equations. For example, to capture the notion of a valid state, a predicate $Inv \subseteq A$ may be used (in place of an initialisation function) to specify the set of all states that the system may exhibit. Also, if $\mathbb{N} \to A$ denotes the set of all possible behaviours (i.e. state streams—maps from discrete time $\mathbb{N}$ to a state-space A) then a predicate $P \subseteq (\mathbb{N} \to A)$ could specify the set of valid behaviours for the given device. Such an approach has the advantage of being able to specify non-deterministic systems i.e. those for which the next state of the system need not be a function of the current state. Temporal abstraction can also be modelled using predicates. For example, if $P \subseteq \mathbb{N}$ is a predicate that identifies 'times of interest' then, by progressively counting the number of times P is true, one can re-number a state stream $Q \subseteq (\mathbb{N} \to A)$ to eliminate all the states that are of no interest—this new (sampled) stream is a temporal abstraction of the original. This report takes a slightly different approach to temporal abstraction. Times of interest are identified using duration functions, which, when given the current state, specify the number of states to be 'skipped' before the next state of interest arises. This is possible when the system under consideration is deterministic, and the required temporal abstraction is strongly state dependent.

By restricting the use of predicates and giving an *operational semantics*, specifications can be readily executed, for example, to simulate the execution of machine code program. Moreover this approach has the advantage that verification proofs are conceptually simple, being founded in first-order equational logic. This helps in terms of the level of human expertise required to understand and carry out proofs, but it also means that a wider range of proof tools can be used.

By using the HOL proof tool, the approach to verification is primarily based around symbolic execution, as opposed to pure model checking, although hybrid approaches are not ruled out. A number of one-step theorems are proved, and these demonstrate that, although time is a useful tool in the specification of processors, correctness proofs need not explicitly involve induction over time. This is achieved by verifying that correctness holds at just two times, $t = 0$ and $t = 1$, where t is the time at the level of an abstract specification. This is possible when initialisation functions behave like state invariants. That is to say, if the range of init is an invariant (specifying all the states that can occur) then it is never necessary to consider the state of the system at times $t > 1$ because all pairs of consecutive states (a_t, a_{t+1}) correspond with some pair $(a_0 = \mathit{init}\, a, a_1 = \mathit{next}\, a_0)$ by virtue of the initialisation function's range and as a consequence of determinism.[1] The proof effort is, therefore, dominated by state-based case analysis, equational reasoning and term-rewriting. As such, much of the proof effort can be carried out automatically by the proof tool. This framework ensures that the approach to the specification and verification of different designs is well-founded and has continuity i.e. follows exactly the same lines at the top-most level.

Section 3 describes an approach to modelling the correctness of conventional processor designs. This is then extended, in Section 4, to take into consideration the temporal behaviour of superscalar designs. Section 5 introduces the mathematical tools required to ensure that correctness proofs are free from induction over time. This work is then explained with a small but complete case-study in Section 6.

2 Related Work

Recent work on microprocessor verification has focused on pipelined and superscalar designs, for example: Tahar and Kumar [28] using HOL, Burch and Dill [3, 2] using model checking; Skakkebæk, Jones and Dill [26] using the Stanford Validity Checker (SVC); Hosabettu, Srivas and Gopalakrishnan [17], and Cyrluk [6] using PVS; and Sawada and Hunt [25] using ACL2 [20]. Particular attention has been paid to managing the complexities associated with such designs, for example, out-of-order issue and interrupts. Topics addressed include, decomposing verifications, developing conducive data/system abstractions, refinements and invariants.

It is worth noting that the models of correctness presented in this report are not necessarily equivalent to those used elsewhere. In particular, none of the approaches listed above use a formal model of temporal abstraction. This report focuses on producing generic models for the study of state systems: these models must be widely applicable and essentially fundamental. Strongly implementation dependent verification methodologies are not examined in any detail within this report.

Other significant work on microprocessor specification and verification includes: work on the SECD chip by Graham and Birtwistle [14]; the verification of the AAMP5 avionics processor by Srivas and Miller [27]; the FM8501 processor verification by Hunt [19]; work on the VIPER microprocessor (reviewed in [1]); Windley's work on Generic Interpreters [31] and the correctness of pipelined processors with Coe [32]; Gordon's work with HOL and LCF-LSM, for example [11, 12]; and Melham's work on abstraction mechanism for hardware verification [22, 23].

---

[1] This example is a slight simplification of the full account because it does not take into consideration the effects of temporal abstraction.

The algebraic approach to modelling microprocessors, as presented and implemented here, was originally developed at the University of Wales Swansea. This work has its foundations in Algebraic Specification and Universal Algebra (for overviews, see [33, 21]) and the study of Synchronous Concurrent Algorithms [29, 30]. Work on microprocessors includes [16], which looked at a microprogrammed example based on Gordon's computer [12], and [9, 10] which looked at superscalar correctness models. A simple pipelined processor was verified in [15] using Maude [5]. The ground work for this report can be found in the thesis [7], which also explains how models with input and output can be supported within this framework.

HOL is founded on Church's theory of simple types [4], and has its origins in Edinburgh LCF [13] and Cambridge LCF [24]. The version of HOL used in the production of this report is HOL98 Taupo-5, which is written in Standard ML (specifically MoscowML). The current HOL distribution, and additional information, may be found at `www.cl.cam.ac.uk/Research/HVG/HOL`. The source for the HOL theories developed in this report may be found at `www.cl.cam.ac.uk/~acjf3`.

3 Algebraic Models

This section introduces modelling techniques suitable for a wide range of processor designs. State systems are considered in Section 3.1, temporal and data abstraction are covered in Sections 3.2 and 3.3 respectively, and correctness is defined in Section 3.4. The approach to temporal abstraction, and consequently correctness, is re-examined in Section 4.

3.1 State Functions and Iterated Maps

A *state function* is any $G : T \to A \to A$ where $T = \{0, 1, \dots\}$ is a set of *cycles* (called a *clock*), and A is any non-empty set or *state-space*. The state $(G\ t\ a)$ represents the state of the system at time t given a preliminary state a. Time is considered in a purely mathematical sense, each cycle need not last for the same amount of *time* as measured in seconds. One cycle (change of state) could take an hour and the next could take a small fraction of a second—this is not specified.

State systems will be modelled deterministically by means of *iterated maps*; these construct sequences of the form

$$\mathit{init}\ a,\ \ \mathit{next}(\mathit{init}\ a),\ \ \mathit{next}(\mathit{next}(\mathit{init}\ a)), \dots, \mathit{next}^t(\mathit{init}\ a), \dots$$

for any given state a. Preliminary states are transformed into initial states using the *initialisation function* init and all subsequent states are generated by repeated application of the *next-state function* next. The system is deterministic because each successive state is a function of the previous state.

An iterated map $G : T \to A \to A$, with initialisation function $\mathit{init} : A \to A$ and next-state function $\mathit{next} : A \to A$, is defined by the equations

$$\begin{aligned} G\ 0\ a &= \mathit{init}\ a, \\ G\ (t+1)\ a &= \mathit{next}(G\ t\ a) \end{aligned}$$

Both init and next are required to be primitive recursive; this ensures that G is a total function over T and A. This equational style of presentation is founded in Algebraic Specification, see [33]. Such an approach has the advantage that proof tools, such as HOL, can be used to automatically evaluate ground terms using first-order reasoning. Although *curried* functions are used in this report (which are not first-order), in each case, *cartesian* versions can be considered if necessary, for example, a state function can be presented as $G : T \times A \to A$.

Iterated maps allow one to specify synchronous, deterministic state systems and have been found particularly suited to modelling the operational semantics of microprocessor architectures and micro-architectures (organisations). These levels of abstraction are more amenable to deterministic approaches because it is not the intent to be wholly true to the physical behaviour of wires and circuitry (where non-deterministic models might be more appropriate). Note that although state transition is synchronous (i.e. clocked), this does not prevent one from specifying processors with clock-less or asynchronous implementations; it is just that, at the level of temporal abstraction modelled, the device behaves synchronously. By considering multiple levels of abstraction one is able to carry out reasoning at levels suitable for the problems at hand. At the level of the micro-architecture (or organisation) one is primarily concerned with the overall functional correctness of a design (be it, microprogrammed, pipelined or superscalar) with respect to the architecture. This level may be modelled in a style significantly more abstract than a register-transfer level description, wherein one would be making more concrete decisions about timing and control signals.

The predicate `IMAP G init next` is defined in HOL to express that `G:num→'a→'a` is the iterated map with initialisation function `init:'a→'a` and next-state function `next:'a→'a`.

```
                                   IMAP_def
⊢ ∀G init next.
      IMAP G init next =
      (∀a. G 0 a = init a) ∧ ∀t a. G (SUC t) a = next (G t a)
```

Type variables (place holders for concrete types) are preceded with a quote, for example, `'a` is used to represent any state-space. The HOL type `num` is used instead of creating a clock datatype from scratch. The natural numbers are a suitable model for a clock, and since extensive theories for the natural numbers have already been developed in HOL, it is judicious to take this route. This mean that no distinction is made, in HOL, between the types of clocks at different levels of abstraction.

State functions and iterated maps with I/O can also be considered, although it is not done so here. This allows one to modularise and compose specifications, thus providing an abstraction mechanism that accommodates a degree of non-determinism with respect to state (by means of input streams) and covers entities that would not form part of a processor's state-space; for example, interrupts, main memory (in the case when one only wishes to model the processor's cache), co-processors and other devices.

3.2 State-Dependent Temporal Abstraction

Temporal abstraction is used as a mechanism for hiding the rate of state transition. This section introduces models of temporal abstraction for circumstances in which each state of a slower system corresponds with one or more states of a faster system. In such cases one system is said to be consistently faster than the other, and temporal abstraction may be likened to sampling the faster clock. This sampling will be modelled using a class of functions called *immersions*. State-dependent temporal abstraction is where the abstraction process is wholly determined by one state of the faster system. The dependence upon a single state arises as a consequence of the fact that the system of interest (i.e. that modelled by a state function) is deterministic and defined with respect to one preliminary state.

An immersion $Imm : A \to T \to S$ is any function defined by, for all $t_1, t_2 \in T$ and $a \in A$

$$Imm\, a\, 0 = 0 \text{ and } t_1 < t_2 \Rightarrow Imm\, a\, t_1 < Imm\, a\, t_2$$

The second condition states that $Imm\, a$ is monotonic. This is sufficient to ensure that clock S is consistently faster than clock T, and consequently, there is never a discrepancy in the ordering of

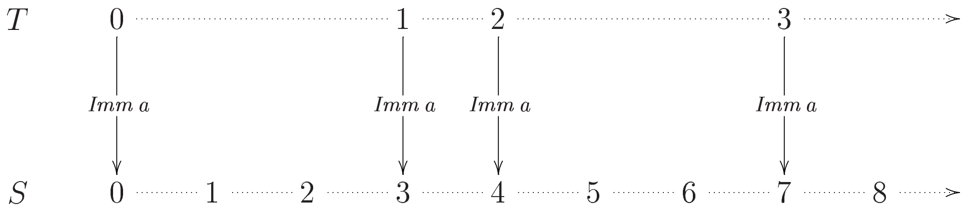

Figure 1: $(Imm\, a) : T \to S$ for state $a \in A$.

events and the abstraction is infinite. Figure 1 shows an immersion that starts by mapping cycles $0, 1, 2$ and 3 on the slow clock T to cycles $0, 3, 4$ and 7 on the faster clock S.

The predicate `IMM Imm` is defined in HOL to express that `Imm:'a->num->num` is an immersion.

`IMM_def`

```
⊢ ∀Imm.
    IMM Imm =
    (∀a. Imm a 0 = 0) ∧ ∀a t1 t2. t1 < t2 ⇒ Imm a t1 < Imm a t2
```

3.3 Data Abstraction

Data abstraction is modelled by means of a map between state-spaces. If $Abs : A \to B$ is a data abstraction from A to B then, in general, one would expect that for each abstract state $b \in B$ there is a concrete state $a \in A$, but this a need not be unique (i.e. Abs is surjective), and hence the map represents an abstraction. In practice the minimum condition placed upon a data abstraction is expressed by the HOL predicate `ONTO_INIT Abs init init'`, which holds when the composition of abstraction and implementation initialisation `(Abs o init'):'a->'b` is surjective with respect to the range of the specification initialisation `init:'b->'b`.

`ONTO_INIT_def`

```
⊢ ∀Abs init init'.
    ONTO_INIT Abs init init' = ∀b. ∃a. Abs (init' a) = init b
```

This condition will be used to ensure that all the initial states of a specification have at least one representative as an initial state in the implementation.

3.4 Implementation Correctness

A state function $Impl : S \to A \to A$ is a *correct implementation* of $Spec : T \to B \to B$ with respect to an immersion $Imm : A \to T \to S$ and data abstraction $Abs : A \to B$ if, and only if, for all $t \in T$ and $a \in A$

$$Spec\, t\ (Abs\ \ a) = Abs\ (Impl\ (Imm\, a\, t)\ a).$$

This correctness condition is illustrated with the following commutative diagram

$$\begin{tikzcd} B \arrow[r, "Spec\, t"] & B \\ A \arrow[u, "Abs"'] \arrow[r, "Impl\, s"] & A \arrow[u, "Abs"'] \end{tikzcd}$$

where $s = Imm\, a\, t$ for all $a \in A$ and $t \in T$.

This shows that by using the data abstraction Abs (which hides the representation of data) and the temporal abstraction Imm (which hides the rate of state change) it is possible to mimic *Spec* via $Impl$. Notice that time is quantified at the level of the specification and state is quantified at the level of the implementation. This has occurred because the immersion Imm is, strictly speaking, carrying out the inverse of temporal abstraction i.e. mapping an abstract clock cycle to a concrete clock cycle. The real abstraction $Ret : A \to S \to T$ is called a *retiming*; but in practice it is easier to define and work with immersions. By the given definition of correctness, the data abstraction relationship (between the states of the implementation and specification) is allowed to break down at cycles between $Imm\,a\ (t+1)$ and $Imm\,a\ t$. This need not matter, provided one is aware that correctness only refers to implementation states occurring at cycles in the range of Imm.

The predicate `CORRECT Spec Impl Imm Abs` is defined in HOL to expresses that the state function `Impl:num→'a→'a` is a correct implementation of the state function `Spec:num→'b→'b` with respect to the immersion `Imm:'a→num→num` and data abstraction `Abs:'a→'b`.

CORRECT_def

```
⊢ ∀Spec Impl Imm Abs.
     CORRECT Spec Impl Imm Abs =
     IMM Imm ∧ ONTO_INIT Abs (Spec 0) (Impl 0) ∧
     ∀t a. Spec t (Abs a) = Abs (Impl (Imm a t) a)
```

4 Algebraic Models for Superscalar Processors

In this section the mathematical tools of Sections 3 are extended in order to take into consideration the more complicated form of temporal abstraction that arises when considering the correctness of superscalar processors.

4.1 Adjunct Immersions

Section 3.2 introduced maps called immersions, which enabled one to model temporal abstraction in the cases when one clock is consistently faster than another. Superscalar processors have a more complicated timing behaviour than this, and this means that the definition of correctness contained in Section 3.4 is no longer adequate. Superscalar processors execute instructions in parallel and, therefore, one is no longer able to capture their timing behaviour using a *total* function from the instruction counting clock T to the implementation's clock S, i.e. using an immersion $Imm : A \to T \to S$. Instead a third clock R is considered; this clock is used to enumerate the execution of blocks of instructions. Two immersions are then used to define the relationship between clocks T and S; one immersion maps from R to T and the other maps from R to S. In this way the specification *Spec* can be perceived as a concrete form of an even more abstract specification, where any number of instructions can execute at the same time.

Figure 2 shows how three clocks and two immersions can be used to express the temporal characteristics of a superscalar processor. The figure shows that the processor starts (from state a) by executing two instructions in four machine cycles, and then three instruction in two cycles. The bottom immersion Imm_2 is defined in a similar way to the immersion of a conventional processor. The top immersion Imm_1, referred to as an *adjunct immersion*, is closely related to Imm_2 but rather than specifying the number of machine cycles it specifies the number of instructions executed.

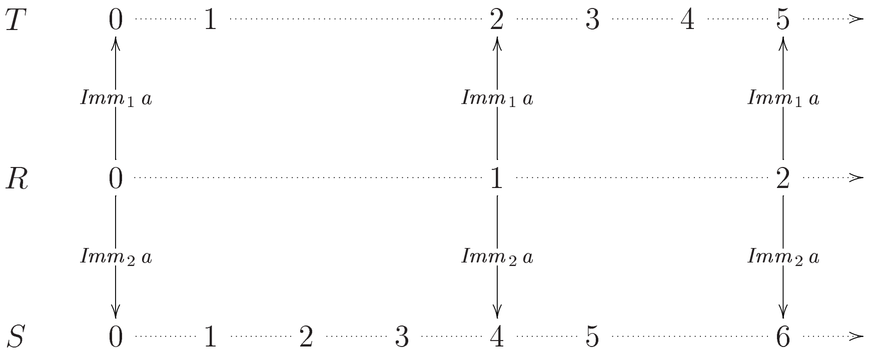

Figure 2: $(Imm_1\ a) : R \to T$ and $(Imm_2\ a) : R \to S$ for state $a \in A$.

4.2 Superscalar Implementation Correctness

A state function $Impl : S \to A \to A$ is a *correct superscalar implementation* of $Spec : T \to B \to B$ with respect to an adjunct immersion $Imm_1 : A \to R \to T$, immersion $Imm_2 : A \to R \to S$ and data abstraction $Abs : A \to B$ if, and only if, for all $r \in R$ and $a \in A$

$$Spec\ (Imm_1\ a\ r)\ (Abs\ \ a) = Abs\ (Impl\ (Imm_2\ a\ r)\ a).$$

This correctness condition is illustrated with the following commutative diagram

$$\begin{tikzcd}
B \arrow[r, "Spec\ t"] & B \\
A \arrow[u, "Abs"] \arrow[r, "Impl\ s"] & A \arrow[u, "Abs"']
\end{tikzcd}$$

where $t = Imm_1\ a\ r$ and $s = Imm_2\ a\ r$ for all $a \in A$ and $r \in R$.

The predicate `CORRECT_SUP Spec Impl Imm1 Imm2 Abs` is defined in HOL to expresses that the state function `Impl:num->'a->'a` is a correct implementation of `Spec:num->'b->'b` with respect to the adjunct immersion `Imm1:'a->num->num`, immersion `Imm2:'a->num->num` and data abstraction `Abs:'a->'b`.

CORRECT_SUP_def

```
⊢ ∀Spec Impl Imm1 Imm2 Abs.
     CORRECT_SUP Spec Impl Imm1 Imm2 Abs =
     IMM Imm1 ∧ IMM Imm2 ∧ ONTO_INIT Abs (Spec 0) (Impl 0) ∧
     ∀r a. Spec (Imm1 a r) (Abs a) = Abs (Impl (Imm2 a r) a)
```

5 One-Step Theorems

This section will focus on the verification of implementation correctness, as defined in Sections 3.4 and 4.2. It will be shown that, by imposing conditions upon the construction of state functions and immersions, it is possible to simplify the correctness condition to just considering the cases $t = 0$ and $t = 1$; thus reducing verifications to case-analysis over the state-space. These results are called one-step theorems.

$$\begin{tikzcd}
T & 0 \arrow[r, dotted, dash] \arrow[d, "Imm\ a"] & 1 \arrow[r, dotted, dash] \arrow[d, "Imm\ a"] & 2 \arrow[r, dotted] \arrow[d, "Imm\ a"] & {} \\
S & 0 \arrow[r, dotted, dash] & dur\ a_{\bar{0}} \arrow[r, dotted, dash] & dur\ a_{\bar{0}} + dur\ a_{\bar{1}} \arrow[r, dotted] & {}
\end{tikzcd}$$

Figure 3: Uniform immersion $Imm : A \to T \to S$ with respect to the maps $G : S \to A \to A$ and $dur : A \to S^{+}$, where $a_{\bar{t}} = G\ (Imm\ a\ t)\ a$.

5.1 Uniform Temporal Abstraction

A *uniform immersion* is an immersion constructed using a state function and a *duration* function, which maps states to clock cycles. Uniform immersions have some desirable properties: they provide a mechanism for connecting state evolution, data abstraction and temporal abstraction, and they are required for the one-step theorems.

Given a state function $G : S \to A \to A$ and a function $dur : A \to S^{+}$, the uniform immersion $Imm : A \to T \to S$ with respect to G and dur is defined by the equations

$$\begin{aligned} Imm\ a\ 0 &= 0, \\ Imm\ a\ (t+1) &= dur(G\ (Imm\ a\ t)\ a) + Imm\ a\ t. \end{aligned}$$

The expansion of this definition, for two cycles, is illustrated in Figure 3. Observe that the interval $Imm\ a\ (t+1) - Imm\ a\ t$ is a function of the implementation state at time t, thus a recurrent state will induce the same temporal abstraction and the immersion will appear to be, in some sense, uniform.

In most cases the temporal characteristics of an implementation (for example, the number of cycles required to execute an instruction) can be expressed as a function of the state of the implementation. Uniform immersions can provide a link with data abstraction because the concept of executing an instruction may be expressed in terms of data abstraction, i.e. the next instruction is executed when the abstracted state of the implementation changes in some prescribed way.

Example. Let state function $F_1 : T \to \mathbb{N} \to \mathbb{N}$ and duration function $dur_1 : \mathbb{N} \to S$ be defined by the equations:

$$\begin{aligned} F_1\ t\ n &= \bar{t} + n, \\ dur_1\ n &= \begin{cases} 1, & \text{if } n \text{ is even} \\ 2, & \text{otherwise;} \end{cases} \end{aligned}$$

where $\bar{\cdot} : T \to \mathbb{N}$ gives the natural number of a clock cycle. Furthermore, let $F_2 : T \to \mathbb{N} \to \mathbb{N}$ and $dur_2 : \mathbb{N} \to S$ be defined by:

$$\begin{aligned} F_2\ t\ n &= (\bar{t} + n) \bmod 2, \\ dur_2\ n &= \tilde{n} + 1 \end{aligned}$$

where $\tilde{\cdot} : \mathbb{N} \to S$ gives the S clock cycle of a natural number.

The uniform immersion $Imm : \mathbb{N} \to T \to S$ with respect to F_1 and dur_1 is also the uniform immersion with respect to F_2 and dur_2. There are two possible temporal abstractions represented by *Imm* n, these being, cycles $0, 1, 2, 3, 4, 5, \ldots$ map to:

$$\begin{array}{ll} 0, 1, 3, 4, 6, 7, \ldots & \text{if } n \text{ is even, and} \\ 0, 2, 3, 5, 6, 8, \ldots & \text{if } n \text{ is odd.} \end{array}$$

This illustrates an important characteristic of uniform immersions. The definition of the temporal abstraction is wholly dependent upon state, through the definitions of state and duration functions. If regular state sequences are generated by this pair of functions then the associated temporal abstractions will also be regular. In the case of F_1 and dur_1, the state function is monotonic increasing and the duration function produces alternating durations, and with F_2 and dur_2 the state function produces alternating states from which the monotonic increasing duration function produces alternating durations.

The predicate `UIMM Imm G dur` is defined in HOL to express that `Imm:'a→num→num` is the uniform immersion with respect to `G:num→'a→'a` and `dur:a'→num`.

`UIMM_def`

```
⊢ ∀Imm G dur.
    UIMM Imm G dur =
    (∀a. 0 < dur a) ∧ (∀a. Imm a 0 = 0) ∧
    ∀a t. Imm a (SUC t) = dur (G (Imm a t) a) + Imm a t
```

Notice that a condition is placed upon the range of `dur` to ensure that the immersion is monotonic. This gives the following lemma, stating that uniform immersions are indeed immersions.

`UIMM_IS_IMM_LEMMA`

```
⊢ ∀Imm G dur. UIMM Imm G dur ⇒ IMM Imm
```

This result is relatively straightforward to prove in HOL using theorems from `arithmeticTheory` and `prim_recTheory`.

5.2 Time-Consistency

In the same way that uniform immersions can be viewed as constructive forms of immersions, time-consistency is a condition which ensures that state functions are defined constructively, enabling the state function to be composed without difficulty.

A state function $G : T \to A \to A$ is *time-consistent* if, and only if, for all $t_1, t_2 \in T$ and $a \in A$

$$G\ (t_1 + t_2)\ a = G\ t_1\ (G\ t_2\ a)$$

This is illustrated with the following commutative diagram:

$$\begin{tikzcd} & A & \\ A \arrow[ur, "G\ (t_1+t_2)"] \arrow[rr, "G\ t_2"'] & & A \arrow[ul, "G\ t_1"'] \end{tikzcd}$$

An iterated map is time-consistent if, and only if, its initialisation function is idempotent on all states generated by the state function, that is, for all $t \in T$ and $a \in A$

$$init\ (G\ t\ a) = G\ t\ a.$$

This follows by observing that the two paths from the bottom-left to the top, in the commutative diagram, differ only in an initialisation occurring after t_2 cycles.

Example. Let iterated map $F : T \to \mathbb{N}^3 \to \mathbb{N}^3$ have definition

$$F\ 0\ (n, s, d) = (n, n^2, 2n + 1),$$
$$F\ (t + 1)\ (n, s, d) = f(F\ t\ (n, s, d))$$

where $f(n, s, d) = (n{+}1, s{+}d, d{+}2)$. The iterated map F is time-consistent because the next-state function f maintains the relationship between n, s and d that is established by the initialisation function $(F\ 0)$.

If $F\ 0\ (n, s, d) = (n, s, d)$ then F would also be time-consistent, but it would have a markedly different behaviour. The next-state function is monotonic, therefore, for F to be time-consistent the initialisation function must not set any of the components to a constant value. Furthermore, if $F\ 0\ (n, s, d) = (n, n^2, 2n)$ then time-consistency would be lost because, after applying the next-state function, s would fail to equal $(n + 1)^2$.

The time-consistency property arises (for the one-step simplification of verification) because the correctness criteria of Section 3.4 relies on the presence of initialisation functions instead of explicitly using an invariant. The implementation's initialisation function *init* is expected to generate the necessary concrete states. If initialisation is too weak (i.e. has too large a range) then correctness may not hold, and if it is too strong (i.e. has too small a range) then (*i*) the property `ONTO_INIT` may not hold, and (*ii*) time-consistency may not hold and one cannot, for example, use the one-step theorem in Section 5.5. Time-consistency ensures that, as well as producing a know valid initial state (for example, by resetting a pipeline) it also generates all the states that occur at cycle $t = 1$ (for example, including those for full and partially full pipelines). Thus, time-consistency requires *init* to do the job of an invariant, and consequently defining an initialisation function for a time-consistent iterated map is non-trivial for complex microprocessor designs such as superscalar micro-architectures.

The predicate `TCON G` is defined in HOL to express that the state function `G:num→'a→'a` is time-consistent.

TCON_def

```
⊢ ∀G. TCON G = ∀t1 t2 a. G (t1 + t2) a = G t1 (G t2 a)
```

The following result (proved using HOL) shows that time-consistency implies that the state function G has an iterated map specification, i.e. it is deterministic. This is easy to prove once one observes that the specification is: `init = (G 0)` and `next = (G 1)`.

TC_IMP_IMAP

```
⊢ ∀G. TCON G ⇒ ∃init next. IMAP G init next
```

The following result (proved in HOL) shows that the time-consistency of an iterated map can be expressed as a constraint on the behaviour of the initialisation function.

TC_LEMMA

```
⊢ ∀G init next. IMAP G init next ⇒ (TCON G = ∀t a. init (G t a) = G t a)
```

The following result (proved in HOL) shows that the identity combinator `I`, when used as an initialisation function, guarantees time-consistency.

TC_I_LEMMA

```
⊢ ∀G next. IMAP G I next ⇒ TCON G
```

5.3 Time-Consistency with an Immersion

A state function $G : S \to A \to A$ is time-consistent with respect to an immersion $Imm : A \to T \to S$ if, and only if, for all $t_1, t_2 \in T$ and $a \in A$

$$G\ (Imm\ (G\ (Imm\ a\ t_2)\ a)\ t_1 + Imm\ a\ t_2)\ a = G\ (Imm\ (G\ (Imm\ a\ t_2)\ a)\ t_1)\ (G\ (Imm\ a\ t_2)\ a).$$

This definition may appear unwieldy but the concept is the same as conventional time-consistency. The left-hand side corresponds with a single application of the state function and the right-hand side corresponds with two application of the state function. The state function is not required to be time-consistent on all cycles $s \in S$, only those for which $s = Imm\ a\ t$ for some $t \in T$ and state $a \in A$. A good example of this is to be found in Section 6.

The predicate `TCON_IMM G Imm` is defined in HOL to express that the function `G:num→'a→'a` is time-consistent with respect to the immersion `Imm:'a→num→num`.

`TCON_IMM_def`

```
⊢ ∀G Imm.
     TCON_IMM G Imm =
     ∀t1 t2 a.
       G (Imm (G (Imm a t2) a) t1 + Imm a t2) a =
       G (Imm (G (Imm a t2) a) t1) (G (Imm a t2) a)
```

The following result is trivial to prove and shows that time-consistency implies time-consistency with respect to any immersion.

`TC_IMP_TC_IMM`

```
⊢ ∀G. TCON G ⇒ ∀Imm. TCON_IMM G Imm
```

The following result (proved in HOL) shows that time-consistency with respect to an immersion effectively places a constraint on the initialisation function at times given by the immersion.

`TC_IMM_LEMMA`

```
⊢ ∀G init next Imm.
     IMM Imm ∧ IMAP G init next ⇒
     (TCON_IMM G Imm = ∀t a. init (G (Imm a t) a) = G (Imm a t) a)
```

5.4 One-Step Time-Consistency Verification

This section introduces two results which show that the time-consistency of an iterated map can be verified by analysis of the cases $t = 0$ and $t = 1$. In the case of time-consistency with respect to an immersion, the result follows when the immersion is uniform.

The HOL theorem `TC_ONE_STEP_THM` proves that the time-consistency of an iterated map specification reduces to proving the idempotency of initialisation at cycles 0 and 1.

`TC_ONE_STEP_THM`

```
⊢ ∀G init next.
     IMAP G init next ⇒
     (TCON G = (∀a. init (G 0 a) = G 0 a) ∧ ∀a. init (G 1 a) = G 1 a)
```

This result is extended with the HOL theorem `TC_IMM_ONE_STEP_THM`, which proves the same result for time-consistency with respect to a uniform immersion.

`TC_IMM_ONE_STEP_THM`

```
⊢ ∀G init next Imm dur.
     IMAP G init next ∧ UIMM Imm G dur ⇒
     (TCON_IMM G Imm =
      (∀a. init (G (Imm a 0) a) = G (Imm a 0) a) ∧
      ∀a. init (G (Imm a 1) a) = G (Imm a 1) a)
```

These two results are proved by induction over the abstract clock and using the results `TC_LEMMA` and `TC_IMM_LEMMA` respectively.

5.5 One-Step Correctness Verification

This section introduces a one-step theorem for the verification of correctness. This theorem states that provided:

- an immersion *Imm* is uniform with respect to an implementation *Impl* and some duration function dur; and
- both *Impl* and the specification *Spec* are time-consistent iterated maps (*Impl* need only be time-consistent with respect to the immersion),

then the correctness of *Impl* is proved if, for all $a \in A$

$$\begin{aligned} \mathit{Spec}\,0\,(Abs\;\;a) &= Abs\;(\mathit{Impl}\;(\mathit{Imm}\,a\,0)\;a), \\ \mathit{Spec}\,1\,(Abs\;\;a) &= Abs\;(\mathit{Impl}\;(\mathit{Imm}\,a\,1)\;a) \end{aligned}$$

where Abs is the data abstraction. This theorem provides for a systematic approach to the verification of correctness. One must first prove that the necessary uniformity and time-consistency conditions hold[2] and then proceed by expanding the definitions of *Spec* and *Impl* using case-analysis over the state-space A.

The theorem `ONE_STEP_THM` is proved in HOL.

`ONE_STEP_THM`

```
⊢ ∀Spec Impl initf nextf initg nextg Imm dur Abs.
     UIMM Imm Impl dur ∧ ONTO_INIT Abs initf initg ∧
     IMAP Spec initf nextf ∧ IMAP Impl initg nextg ∧ TCON Spec ∧
     TCON_IMM Impl Imm ⇒
     (CORRECT Spec Impl Imm Abs =
      (∀a. Spec 0 (Abs a) = Abs (Impl (Imm a 0) a)) ∧
      ∀a. Spec 1 (Abs a) = Abs (Impl (Imm a 1) a))
```

5.6 Uniform Adjunct Immersions

In order to prove a one-step theorem for superscalar implementation correctness, one must first consider the construction of adjunct immersions. A uniform adjunct immersion is constructed using a state function, duration function and a regular uniform immersion.

Given a state function $G : S \to A \to A$, a function $dur_1 : A \to T^+$ and a uniform immersion $\mathit{Imm}_2 : A \to R \to S$ (defined with respect to G and some $dur_2 : A \to S^+$), then the *uniform*

---

[2] Uniformity should follow from the definition of the immersion. Time-consistency may be quite difficult to prove (i.e. involve a great deal of case-analysis) but can be done using the theorems `TC_ONE_STEP_THM` and `TC_IMM_ONE_STEP_THM`.

adjunct immersion $Imm_1 : A \to R \to T$ with respect to G, dur_1 and Imm_2 is defined by the equations

$$\begin{aligned} Imm_1\, a\, 0 &= 0, \\ Imm_1\, a\, (r+1) &= dur_1(G\,(Imm_2\, a\, r)\, a) + Imm_1\, a\, r. \end{aligned}$$

This immersion differs in construction from a regular uniform immersion because the state of the implementation at cycle r is not determined by Imm_1 itself, but is given by the predefined uniform immersion Imm_2.

The HOL predicate `AUIMM Imm1 Imm2 G dur1 dur2` is used to express that the immersion `Imm2:'a→num→num` is uniform with respect to state function `G:num→'a→'a` and duration function `dur2:'a→num`, and that `Imm1:'a→num→num` is a uniform adjunct immersion with respect to `G`, `dur1:'a→num` and `Imm2`.

AUIMM_def

```
⊢ ∀Imm1 Imm2 G dur1 dur2.
     AUIMM Imm1 Imm2 G dur1 dur2 =
     UIMM Imm2 G dur2 ∧ (∀a. 0 < dur1 a) ∧ (∀a. Imm1 a 0 = 0) ∧
     ∀a t. Imm1 a (SUC t) = dur1 (G (Imm2 a t) a) + Imm1 a t
```

The lemma `AUIMM_IS_IMM_LEMMA` shows that `Imm1` and `Imm2`, as defined above, are both immersions.

AUIMM_IS_IMM_LEMMA

```
⊢ ∀G Imm1 Imm2 dur1 dur2.
     AUIMM Imm1 Imm2 G dur1 dur2 ⇒ IMM Imm1 ∧ IMM Imm2
```

5.7 One-Step Superscalar Correctness Verification

The theorem `ONE_STEP_SUP_THM` is proved in HOL and is a superscalar variant of the theorem `ONE_STEP_THM`.

ONE_STEP_SUP_THM

```
⊢ ∀Spec Impl initf nextf initg nextg Imm1 Imm2 dur1 dur2 Psi.
     AUIMM Imm1 Imm2 Impl dur1 dur2 ∧ ONTO_INIT Psi initf initg ∧
     IMAP Spec initf nextf ∧ IMAP Impl initg nextg ∧ TCON Spec ∧
     TCON_IMM Impl Imm2 ⇒
     (CORRECT_SUP Spec Impl Imm1 Imm2 Psi =
      (∀a. Spec (Imm1 a 0) (Psi a) = Psi (Impl (Imm2 a 0) a)) ∧
      ∀a. Spec (Imm1 a 1) (Psi a) = Psi (Impl (Imm2 a 1) a))
```

It is worth noting that the time-consistency of *Spec* is not expressed in the more general setting i.e. with respect to the immersion Imm_1. The reason for this is that the state function for an architecture will invariably be fully time-consistent because it is possible to execute (specify) each instruction one at a time.

6 Example: A Datapath with Microprogram Control

The methods and techniques of Sections 3 and 5 are demonstrated with a relatively simple, but complete, example. Section 6.1 gives a Programmer's Model (PM) specification of a device, which consists of: a read-only memory, a counter and an accumulator. An implementation of this device is then specified in Section 6.2. The implementation is modelled as an Abstract Circuit (AC), employing a single-bus datapath with microprogram control. The correctness of the AC, with respect to the PM, is defined in Section 6.3, and this is then verified in Section 6.4. Although the device is not as complex as a microprocessor, the methods used (both in the specification and verification of the AC) are equally suited to the verification of a microprocessor with microprogram control logic.

6.1 PM Specification

In this section a simple device is formally specified using HOL. The device has three components: a read-only memory `m`, a counter `pc` and an accumulator `acc`. The memory is indexed by the counter and each address contains a single bit of data. The intended Programmer's Model (PM) behaviour of the device is as follows:

> *If the bit* `m(pc)` *is set then the counter is added to accumulator* `acc:=acc+pc`*, otherwise a no-op (no-operation) occurs. In both cases, the counter is incremented and the device continues in this manner ad infinitum.*

The state-space and underlying operations are under-specified. That is to say, the counter and accumulator could range over the natural numbers, with the memory being infinite. On the other hand, they could range over a single bit or bit-vector—this would mean that the state-space is finite, registers would overflow and the behaviour of the device would be cyclic (repeat the same sequence of states). In either case, the device does not have any obvious purpose, beyond that which it is put to here.

The state-space of the device is specified in HOL with the following type declarations.

```
new_type {Arity = 0, Name = "word"};
Hol_datatype ‘state_PM = PM of (word->bool)=>word=>word‘;
```

The state-space is represented by the type `state_PM`. The type `word` is declared, together with the operations `ADD:word→word→word` and `INC:word→word`, but no attempt is made to ascribe a semantics. The memory is modelled as a map from `word` to `bool`.

The PM state function `STATE_PM:num→state_PM→state_PM` is defined as follows:

STATE_PM_def

```
⊢ (∀a. STATE_PM 0 a = a) ∧
   ∀t a. STATE_PM (SUC t) a = NEXT_PM (STATE_PM t a)
```

The PM next-state function `STATE_PM:state_PM→state_PM` is defined as follows:

NEXT_PM_def

```
⊢ NEXT_PM (PM m pc acc) =
   (if m pc then PM m (INC pc) (acc ADD pc) else PM m (INC pc) acc)
```

There are two cases, one of which is a no-op.

6.2 AC Specification

The PM device is implemented as an Abstract Circuit (AC), which contains a single-bus datapath and employs microprogram control, see Figure 4. The bus is capable of carrying one word at a time and the Arithmetic-Logic Unit (ALU) can perform one addition (or increment) per cycle. Consequently, more than one cycle is required for the AC to implement the functionality of the PM. The

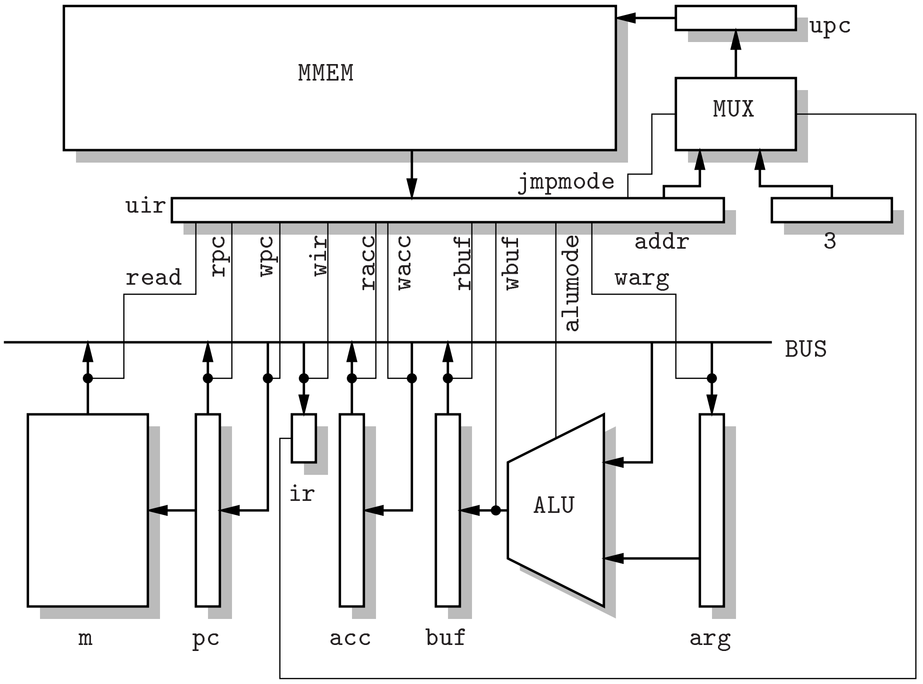

Figure 4: The AC datapath and microprogram control.

Line	Micro-instruction		Comment
0	`read, wir`	; 1	Fetch instruction
1	`jmpmode`	; `ir` ⇒ 3 | 2	Decode instruction
2		; 6	No-op
3	`racc, warg`	; 4	Read accumulator
4	`rpc, add`	; 5	. . . perform addition with `pc`
5	`rbuf, wacc`	; 6	. . . store result in `acc`
6	`rpc, inc`	; 7	Increment `pc`
7	`rbuf, wpc`	; 0	. . . store result in `pc`

Table 1: The microprogram `MMEM`. The mnemonic `add` is used to indicate that both `wbuf` and `alumode` are set, and `inc` is a pseudonym for `wbuf`. The value after the semi-colon gives the next micro-instruction address. If `jmpmode` is set then the next line is determined by the register `ir`, see Figure 4.

datapath is controlled by a series of single-bit control flags, contained in a 14-bit micro-instruction `uir`. The overall behaviour of the AC is controlled by the contents of the microprogram memory `MMEM`, from which the current micro-instruction fetched using a microprogram-counter `upc`. An annotated description of the microprogram `MMEM` is provided in Table 1. The implementation is somewhat artificial; for example, it would be more sensible to dispense with the `arg` register, connecting the `pc` register directly to the ALU. The given arrangement has been chosen because it provides a fair representation of the datapath of microprogrammed microprocessor.

The state-space of the AC is specified in HOL with the following type declarations.

```
Hol_datatype ‘upc = MPC of bool⇒bool⇒bool‘;
Hol_datatype ‘uir = MIR of bool⇒bool⇒bool⇒bool⇒bool⇒bool⇒bool⇒
                           bool⇒bool⇒bool⇒bool⇒bool⇒bool⇒bool‘;
Hol_datatype ‘datapath = DP of word⇒bool⇒word⇒word⇒word‘;
Hol_datatype ‘state_AC = AC of (word→bool)⇒datapath⇒upc⇒(upc→uir)‘;
```

The microprogram-counter is three bits long and the micro-instruction register is 14 bits long.[3] The datapath `DP pc ir acc buf arg` does not include the read-only memory.

The AC state function `STATE_AC:num→state_AC→state_AC` is defined as follows:

`STATE_AC_def`

```
⊢ (∀a. STATE_AC 0 a = INIT_AC a) ∧
   ∀t a. STATE_AC (SUC t) a = NEXT_AC (STATE_AC t a)
```

The AC initialisation function `INIT_AC:state_AC→state_AC` is defined as follows:

`INIT_AC_def`

```
⊢ INIT_AC (AC m datapath upc um) = AC m datapath (MPC F F F) MMEM
```

The initialisation function sets the microprogram counter to address zero and the microprogram to the constant `MMEM` (see Figure 5). By treating the microprogram as part of the state-space (as opposed to hard-wiring it in the definition of the next-state function), this emphasises the fact that the device could be correct when initialised with different microprograms.

The AC next-state function `NEXT_AC:state_AC→state_AC` is defined as follows:

`NEXT_AC_def`

```
⊢ NEXT_AC (AC m datapath upc um) =
   (let uir = um upc in
      AC m (BUS_READ uir datapath (BUS_WRITE uir m datapath))
        (NEXT_MPC uir (IR datapath) um)
```

The functions `BUS_WRITE`, `BUS_READ` and `NEXT_MPC` are all defined in Figure 5. The function `IR:datapath→bool` simply projects out the instruction register `ir`. One can identify three parts to micro-instruction execution: (*i*) the micro-instruction register `uir` is fetched, to become `um(upc)`, (*ii*) there is a write to the bus, from a source controlled by `uir`, followed by a read from the bus, to a destination controlled by `uir`; and (*iii*) the microprogram counter is updated.

---

[3] In hardware not all micro-instruction words are valid: one must ensure that certain control flags are mutually exclusive (disjoint). For example, attempts to simultaneously write multiple values to the bus must be precluded. This potential problem is not reflected directly in the specification of AC, wherein nested-if structures effectively prioritises bus read/write control signals. Though the microprogram `MMEM` (Figure 5) would not cause a problem in hardware.

```
val BUS_WRITE_def =
    ⊢ ∀uir m pc ir acc buf arg.
         BUS_WRITE uir m (DP pc ir acc buf arg) =
         (if READ uir then
            PAD (m pc)
          else
            (if RPC uir then
               pc
             else
               (if RACC uir then
                  acc
                else
                  (if RBUF uir then buf else PAD ARB))))
val ALU_def =
    ⊢ ∀uir arg bus.
         ALU uir arg bus = (if ALUMODE uir then arg ADD bus else INC bus)

val BUS_READ_def =
    ⊢ ∀uir pc ir acc buf arg bus.
         BUS_READ uir (DP pc ir acc buf arg) bus =
         (if WPC uir then
            DP bus ir acc buf arg
          else
            (if WIR uir then
               DP pc (TRIM bus) acc buf arg
             else
               (if WACC uir then
                  DP pc ir bus buf arg
                else
                  (if WBUF uir then
                     DP pc ir acc (ALU uir arg bus) arg
                   else
                     (if WARG uir then
                        DP pc ir acc buf bus
                      else
                        DP pc ir acc buf arg)))))
val NEXT_MPC_def =
    ⊢ ∀uir ir.
         NEXT_MPC uir ir = (if JMPMODE uir ∧ ir then MPC F T T else ADDR uir)

uir = (MIR read rpc wpc wir racc wacc rbuf wbuf warg alumode jmpmode addr2 addr1 addr0)

val MMEM_def =
    ⊢  (MMEM (MPC F F F) = MIR T F F T F F F F F F F F F T) ∧
       (MMEM (MPC F F T) = MIR F F F F F F F F F F T F T F) ∧
       (MMEM (MPC F T F) = MIR F F F F F F F F F F F T T F) ∧
       (MMEM (MPC F T T) = MIR F F F F T F F F T F F T F F) ∧
       (MMEM (MPC T F F) = MIR F T F F F F F T F T F T F T) ∧
       (MMEM (MPC T F T) = MIR F F F F F T T F F F F T T F) ∧
       (MMEM (MPC T T F) = MIR F T F F F F F T F F F T T T) ∧
       (MMEM (MPC T T T) = MIR F F T F F F T F F F F F F F)
```

Figure 5: The main AC functions and the microprogram memory `MMEM`. The twelve projection functions `READ:uir→bool`, `RPC:uir→bool` and so forth, have obvious definitions. In the case of the single bit register `ir`, the maps `PAD` and `TRIM` are used when writing to, and reading from, the bus.

6.3 Correctness Statement

The correctness of the AC is expressed in HOL using the predicate `CORRECT` from page 7.

`CORRECT_AC_def`

```
⊢ CORRECT STATE_PM STATE_AC IMM_AC ABS_AC
```

The data abstraction `ABS_AC:state_AC→state_PM` is defined as follows:

`ABS_AC_def`

```
⊢ ABS_AC (AC m (DP pc ir acc buf arg) upc um) = PM m pc acc
```

The abstraction function projects out the memory, counter and accumulator from the AC state. In many cases, data abstractions will be based around projection. Though care must be taken with pipelined designs, where the AC program-counter is allowed to run ahead of the PM program-counter.

The immersion `IMM_AC:state_AC→num→num`, defined below, is uniform with respect to the state function `STATE_AC` and duration function `DUR_AC`.

`IMM_AC_def`

```
⊢ (∀a. IMM_AC a 0 = 0) ∧
   ∀a t. IMM_AC a (SUC t) = DUR_AC (STATE_AC (IMM_AC a t) a) + IMM_AC a t
```

The duration function `DUR_AC:state_AC→num` is defined as follows:

`DUR_AC_def`

```
⊢ DUR_AC (AC m (DP pc ir acc buf arg) upc um) = (if m pc then 7 else 5)
```

By examining the microprogram `MMEM` (Table 1), one can observe that there are only two possible program traces: lines 0, 1, 3, 4, 5, 6, 7, 0 if `m(pc)` is set, and lines 0, 1, 2, 6, 7, 0 otherwise. Hence the PM instruction execution is complete (`upc` equals zero) at cycles seven and five respectively.

Clearly microprocessors will have more complex duration functions, and one might consider using a bounded search to establish when instructions have been *committed*. In many cases though, the temporal characteristics are remarkably easy to deduce because this information is reflected in the control logic. Certain classes of instructions and events (such as interrupts) will have know temporal characteristics.

6.4 Verification

The theorem `ONE_STEP_THM` (page 13) is used to verify the correctness of the AC. Before applying this theorem to the correctness statement, a number of conditions must be satisfied. One must prove that: both of the state functions are iterated maps; the immersion is uniform; the `ONTO_INIT` property holds (page 6); and both state functions are time-consistent.

6.4.1 Preliminaries

The theorem `STATE_PM_THM` confirms that `STATE_PM` is an iterated map, with the identity map `I` as the initialisation function and `NEXT_PM` as the next-state function.

`STATE_PM_THM`

```
⊢ IMAP STATE_PM I NEXT_PM
```

The theorem `STATE_AC_THM` shows that `STATE_AC` is an iterated map with initialisation function `INIT_AC` and next-state function `NEXT_AC`.

STATE_AC_THM

```
⊢ IMAP STATE_AC INIT_AC NEXT_AC
```

Both of the theorems above follow from the construction of the state functions and have one line proofs in HOL.

The following theorem shows that the immersion `IMM_AC` is uniform.

IMM_AC_UNIFORM

```
⊢ UIMM IMM_AC STATE_AC DUR_AC
```

This result follows from the construction of `IMM_AC`. During the proof it is necessary to prove that zero is not in the range of the duration function `DUR_AC`. This may be established by trivial case-analysis: there are only two possible durations (seven and five cycles) both of which are non zero.

The result `AC_ONTO_INIT` establishes that all the initial PM states are represented by a concrete initial state in the AC.

AC_ONTO_INIT

```
⊢ ONTO_INIT ABS_AC I INIT_AC
```

This result is proved in HOL using `EXISTS_TAC` to give witness to the necessary implementation states: components that `ABS_AC` projects out (to form the specification state) are set appropriately, and all other values are instantiated with `ARB=εx.T` to denote arbitrary values.

6.4.2 Time-Consistency

Having established that `STATE_AC` is an iterated map and `IMM_AC` is a uniform immersion, it is possible to verify the following result using `TC_IMM_ONE_STEP_THM` from page 12.

AC_TCON_IMM

```
⊢ TCON_IMM STATE_AC IMM_AC
```

After applying `TC_IMM_ONE_STEP_THM` to the initial goal, the following two subgoals emerge:

```
INIT_AC (STATE_AC (IMM_AC a 1) a) = STATE_AC (IMM_AC a 1) a
____________________________________________________________

  0.  IMAP STATE_AC INIT_AC NEXT_AC
  1.  UIMM IMM_AC STATE_AC DUR_AC

INIT_AC (STATE_AC (IMM_AC a 0) a) = STATE_AC (IMM_AC a 0) a
____________________________________________________________

  0.  IMAP STATE_AC INIT_AC NEXT_AC
  1.  UIMM IMM_AC STATE_AC DUR_AC
```

The first sub-goal ($t = 0$) is easy to prove: it simply compares two applications of `INIT_AC` with one application. The second sub-goal ($t = 1$) is dealt with by case-analysis using the following lemma.

AC_LEMMA

```
⊢ ∀a. (a = AC m (DP pc ir acc buf arg) upc um) ⇒
     (STATE_AC (IMM_AC a 1) a =
      (if m pc then
         AC m (DP (INC pc) T (acc ADD pc) (INC pc) acc) (MPC F F F) MMEM
       else
         AC m (DP (INC pc) F acc (INC pc) arg) (MPC F F F) MMEM))
```

If `m(pc)` is set then the next-state function `NEXT_AC` is applied for seven cycles, otherwise it is applied for five cycles. In both cases one can observe that the microprogram memory `MMEM` is not altered and the microprogram-counter `upc` is restored to line zero. These are the only two components that could have been altered by the initialisation function, consequently applying `INIT_AC` has no affect and time-consistency is proved.

This lemma is also used in the next section where the correctness of the AC is finally proved. Symbolically evaluating the state function with this lemma prevents repeating the same re-writes and simplifications in different parts of the proof.

This example highlights the fact that the initialisation function `INIT_AC` is being used to check the consistency of the implementation's state, after having applied the next-state function i.e. it is expressing an implicit invariant condition. If the microprogram were changed, or if the microprogram-counter had an unexpected value, then this would have been picked up during the proof for time-consistency. Failure to be time-consistent need not imply incorrect behaviour but it does preclude the use of `ONE_STEP_THM`. If one is confident that the emergent behaviour is correct then one may choose to modify an initialisation function accordingly. Notice that the initialisation function could have been weakened (to cover more cases) without good reason, for example, to consider the cases where the AC was some way through the execution of an microprogram instruction sequence. This would lead to achieving full time-consistency, as given by the predicate `TCON` in Section 5.2. By expressing time-consistency in terms of the immersion `IMM_AC`, it is only necessary to consider states corresponding with the completion of PM instructions.

`AC_LEMMA` is proved using a single application of the HOL tactic `RW_TAC`. The following simpset is used:

`exec_ss`

```
std_ss++rewrites [INIT_AC_def,NEXT_AC_def,NEXT_MPC_def,TRIM_AX,
  BUS_READ_def,BUS_WRITE_def,ALU_def,IR_def,READ_def,RPC_def,WPC_def,
  WIR_def,RACC_def,WACC_def,RBUF_def,WBUF_def,WARG_def,ALUMODE_def,
  JMPMODE_def,ADDR_def,MMEM_def]
```

Term rewriting with this simpset, together with case-splitting, is sufficient to generate all the possible next state cases for the AC. One case-split is introduced for every cycle of the PM clock, and because the one-step theorems are being used, it is only necessary to consider a single cycle. When verifying microprocessors, the number of cases will typically scale with the size of the instruction set, the length of pipelines, and the width of superscalar designs. Finding ways to manage the potentially huge number of state cases inherent in current microprocessor designs (with multiple execute units, interrupts and very long pipelines) is an active research area.

6.4.3 Correctness

All of the apparatus is now available to complete the AC correctness verification. The proof starts by assuming the results established in the last couple of sections. The time-consistency of the specification is then deduced using `TC_I_LEMMA`. After this, the theorem `ONE_STEP_THM` can be used to re-write the proof obligation, thus eliminating the time variable. The state is then expanded, using `Cases_on`, to give all the components of the AC state-space; this expression is abbreviated later using `ABBREV_TAC`.

The $t = 0$ case is readily proved using the simplifier tactic `SIMP_TAC`. This confirms that the AC starts in a valid initial state because `INIT_AC` does not change any of the PM components. The proof is finished off ($t = 1$ case) by calling upon `AC_LEMMA` followed by one application of `RW_TAC`, which is where the next states of the implementation (as defined by the lemma) are compared with those of the specification.

The complete proof is as follows:

```
CORRECT STATE_PM STATE_AC IMM_AC ABS_AC
────────────────────────────────────────────────────────────────

MAP_EVERY ASSUME_TAC [STATE_PM_THM,STATE_AC_THM,IMM_AC_UNIFORM,
                      AC_ONTO_INIT,AC_TIME_CON_IMM]
  THEN ‘TCON STATE_PM‘ by IMP_RES_TAC TC_I_LEMMA
  THEN ‘CORRECT STATE_PM STATE_AC IMM_AC ABS_AC =
         (∀a. STATE_PM 0 (ABS_AC a) = ABS_AC (STATE_AC (IMM_AC a 0) a)) ∧
          ∀a. STATE_PM 1 (ABS_AC a) = ABS_AC (STATE_AC (IMM_AC a 1) a)‘
    by IMP_RES_TAC ONE_STEP_THM
  THEN POP_ASSUM (fn th => REWRITE_TAC [th])
  THEN REPEAT STRIP_TAC
  THEN Cases_on ‘a‘
  THEN Cases_on ‘d‘
  THENL [
    SIMP_TAC std_ss [STATE_PM_def,STATE_AC_def,IMM_AC_def,INIT_AC_def,ABS_AC_def],
    ABBREV_TAC ‘a = AC f (DP w b w0 w1 w2) u f0‘
      THEN POP_ASSUM (fn th ⇒ ASSUME_TAC (SYM th))
      THEN IMP_RES_TAC AC_LEMMA
      THEN RW_TAC std_ss [ONE,STATE_PM_def,NEXT_PM_def,ABS_AC_def]]
```

7 Future Work and Conclusions

This report has established that the HOL proof tool can be used to support an algebraic framework for modelling and verifying microprocessors. This has involved formalising models of data and temporal abstractions, in order to present a precise and general formulation of correctness. It is asserted that this work is applicable to a wide range of deterministic state systems, including abstract models of pipelined and superscalar processors. This material was then explained and demonstrated with a methodical verification of a microprogrammed abstract circuit.

This framework has been used to verify a pipelined design [8]. To take this work forward, a far more sizable case-study will be undertaken. A large part of the ARM instruction set will be specified in HOL, together with a pipelined implementation, loosely based around the ARM6 core. Initially this will establish whether or not such a specification and verification is feasible in HOL, using the methods presented in this report. In the long term it will hopefully provide a basis for looking at other features, for example, the incorporation of caches, interrupts, and multi-processor systems.

References

[1] Bishop C. Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. Technical Report 46, Computational Logic, Inc., Austin, Texas, January 1990.

[2] Jerry R. Burch. Techniques for verifying superscalar microprocessors. In *33rd Design Automation Conference*, pages 552–557. ACM Press, 1996.

[3] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of the 6th International Conference, CAV '94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Berlin, 1994. Springer-Verlag.

[4] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: Specification and programming in rewrite logic. Technical report, Computer Science Laboratory, SRI International, March 1999.

[6] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, 1993.

[7] Anthony C. J. Fox. *Algebraic Models for Advanced Microprocessors*. PhD thesis, University of Wales Swansea, 1998.

[8] Anthony C. J. Fox. The specification and verification of a 4-stage RISC pipeline using HOL. Technical report, Computer Laboratory, University of Cambridge, 2001. In preparation.

[9] Anthony C. J. Fox and Neal A. Harman. An algebraic model of correctness for superscalar microprocessors. In Mandayam K. Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference, FMCAD '96: Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, 1996.

[10] Anthony C. J. Fox and Neal A. Harman. Algebraic models of superscalar microprocessor implementations: A case study. In B. Möller and J V Tucker, editors, *Prospects for Hardware Foundations*, volume Lecture Notes in Computer Science 1546, pages 138–183. Springer-Verlag, 1998.

[11] Mike J. C. Gordon. LCF-LSM: A system for specifying and verifying hardware. Technical Report 41, University of Cambridge Computer Laboratory, 1983.

[12] Mike J. C. Gordon. Proving a computer correct with the LCF-LSM hardware verification system. Technical Report 42, University of Cambridge Computer Laboratory, 1983.

[13] Mike J. C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[14] Brian Graham and Graham Birtwistle. Formalising the design of an SECD chip. In Miriam E. Leeser and Geoffrey M. Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*, pages 40–66. Springer-Verlag, 1990.

[15] Neal A. Harman. Verifying a simple pipelined microprocessor using Maude. Technical Report 4, Department of Computer Science, University of Wales Swansea, 2000.

[16] Neal A. Harman and John V. Tucker. Algebraic models of microprocessors: The verification of a simple computer. In V Stravridou, editor, *Mathematics of Dependable Systems II*, pages 135–170. Oxford University Press, 1997.

[17] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In Hu and Vardi [18], pages 122–134.

[18] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification: 10th International Conference, CAV'98 Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[19] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas at Austin, December 1985.

[20] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[21] Karl Meinke and John V Tucker. Universal algebra. In S Abramsky, D Gabbay, and T S E Maibaum, editors, *Handbook of Logic in Computer Science*, pages 189 – 411. Oxford University Press, 1992.

[22] Thomas F. Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291. Kluwer Academic Publishers, 1988.

[23] Thomas F. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[24] Larry Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume Cambridge Tracts in Theoretical Computer Science 2. Cambridge University Press, 1987.

[25] Jun Sawada and Warren A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In Hu and Vardi [18], pages 135–146.

[26] Jens U. Skakkebæk, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In Hu and Vardi [18], pages 98–109.

[27] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[28] Sofiène Tahar and Ramayya Kumar. A practical methodology for the formal verification of RISC processors. *Formal Methods in Systems Design*, 13(2):159–225, September 1998.

[29] Ben C. Thompson. *A Mathematical Theory of Synchronous Concurrent Algorithms*. PhD thesis, Department of Computer Studies, University of Leeds, 1987.

[30] Ben C. Thompson and John V. Tucker. Equational specification of synchronous concurrent algebras and architectures. Technical Report CSR-9.91, Department of Computer Science, University College Swansea, 1991.

[31] Phillip J. Windley. A theory of generic interpreters. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pages 122–134. Springer-Verlag, 1993.

[32] Phillip. J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Ramayya Kumar and Thomas Kropf, editors, *Proceedings of the 2nd International Conference, TPCD '94: Theorem Provers in Circuit: Theory, Practice and Experience*, volume 901 of *Lecture Notes in Computer Science*, pages 33–51. Springer-Verlag, 1995.

[33] Martin Wirsing. Algebraic specification. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 675 – 788. Elsevier, 1990.