

Algebraic Models of Temporal Abstraction for Initialised Iterated State Systems: An Abstract Pipelined Case Study

A. C. J. Fox and N. A. Harman,
Department of Computer Science,
University of Wales Swansea,
Singleton Park,
Swansea SA2 8PP

1 November 1998

Abstract

The data and temporal abstractions of a pipelined case study are explored in an algebraic setting. We apply a set of algebraic tools for modelling microprocessors to the specification, pipelined implementation, and formal verification of an abstract case study. We employ a model of time based on counting events by means of a *clock*. We model systems as *iterated maps* that evolve over time from some initial state. We define formal correctness conditions, and introduce the *one-step theorems* that, given certain conditions, reduce the complexity of formal verification. The algebraic models provide: (i) modular descriptions of pipelined systems, including correctness criteria; and (ii) equational specification and verification techniques for the design of pipelined systems applicable to a range of specification languages and theorem provers.

1 Introduction

This paper examines the nature of initialisation, data abstraction and temporal abstraction for pipelined systems. This work has its origins in the study of the correctness of pipelined microprocessor organisations [11], and forms part of a series on algebraic models of microprocessors. In [18, 19] simple, *microprogrammed* examples are considered, respectively with and without *input streams*. In [13], *correctness models* for microprocessors are considered, and the formal verification process examined. In [12, 14] *super-scalar* processors are examined by means of a substantial example.

We are particularly interested in models of time, and the complex temporal relationships between microprocessors at different levels of abstraction.

We employ a model of time based on a *clock* that divides time into segments defined by events. We relate different levels of timing abstraction, or clocks, by surjective, monotonic maps called *retimings*. Microprocessors, and related systems, are modelled as *iterated maps* $F : T \times A \rightarrow A$, where T is a clock, dividing up time and A represents the state-set of the microprocessor (registers and memories). F is equationally defined as follows

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

where 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 also acts as an invariant when applying the *one-step theorems* (Sect. 3.7, [13, 11]) to reduce formal verification to state exploration. In this paper, we omit discussion of input and output: however, this is easily accommodated [19, 11]. To illustrate our techniques, we formally verify two simple implementations of a pipelined system. The first implementation contains a pipeline that is always full. The second has a pipeline that may be empty on initialisation.

The algebraic methods on which this work is based: (i) are modular, and provide a basis for the formal decomposition of the descriptions of microprocessors and associated correctness criteria; and (ii) support equational specification and verification techniques for the design of microprocessors, that are not dedicated to specific software systems (term rewriting systems, theorem provers), but are general, and may be represented in, and processed by, a range of machine reasoning systems. In addition, they form the basis of a uniform theoretical framework for modelling microprocessors. Extended discussion of the work in this paper, and of that in [13, 12], can be found in [11].

Much of the work on pipelined microprocessors has been greatly influenced by the use of *software tools* and is motivated by the need to verify *specific* designs using *specific* theorem provers. This paper is primarily concerned with a general theoretical framework for modelling microprocessors; in particular, the notion of *correctness* and the use of *abstraction mechanisms*. We present a case study that is not intended to be wholly representative of existing microprocessor designs. Instead, focus is placed on correctness issues while avoiding all unhelpful and uninformative complexities. It is acknowledged that software tools are essential for the verification of ‘real world’ designs, but it is also important to establish an underlying theory to compliment verifications. Concrete case studies and verification attempts are more fully understood, and readily managed, when viewed in the context of a general, well-established and mathematically rigorous theory.

The structure of this paper is as follows. In Sect. 2 we outline the underlying concepts of our model, and relate it to the work of others. In Sect. 3 we introduce the fundamental algebraic tools for modelling iterated maps, time, timing abstraction, and correctness. In Sect. 4 we introduce an abstract case study, and develop and prove two pipelined implementations (the proofs are banished to the appendix). In Sect. 5 we summarise our techniques and their applicability.

This work was undertaken as part of the Esprit Working Group NADA (00 85 33) on new methods for hardware description languages [31].

2 Evolving State Systems

In this section a philosophical basis for the work of Sect. 3 and Sect. 4 is outlined, and set in the context of related work on modelling pipelined systems.

2.1 States and State Sequences

Models are constructed of systems that exhibit a property called *state*. By defining state, a level of abstraction is established. For example, one may define the state of a traffic signal to be red, yellow or green; but one could choose a more precise description of the light (wavelength and intensity), or observe other properties of the traffic signal such as temperature, location, mass and so forth. In general, we aim to choose those properties of a system related to the key aspects under consideration. In a digital system this will generally be [some abstraction of] registers and memories. Intuitively state is the observed ‘current condition of being’ of the model. This principle of observation is important when considering temporal abstraction.

Computer systems are modelled by considering state-sequences. For example,

$$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \dots$$

where a_i represents the i^{th} state of a system. The *state-space* of the system is the non-empty¹ set of all possible system states. Only deterministic state evolution is considered in this paper. That is, any given state is always followed by the same *next-state*². In the example above, the initial state a_0 is always followed by the state a_1 . By determinism, once a state re-occurs then all subsequent states must re-occur (in the same order) and the state evolution is *cyclic*.

¹It is assumed that the system can exhibit at least one state.

²The absence of non-determinism has, to date, not proved restrictive at the levels of abstraction dealt with in this paper.

2.2 A Philosophy of Time

A formal notion of time is defined by the enumeration of *state change*: time does not exist in itself, but arises from considering the ordering of distinct occurrent states. If a system does not change state, or ceases to change state, then time is, or becomes, redundant; because there is no next-state. For example, consider the following finite state sequence

$$a_0 \rightarrow a_1.$$

By definition the zeroth state a_0 occurs at time zero and the first state a_1 occurs at time one. There is no time two because the system only changes state once. Further, given such a model of time, the state sequence

$$a_0 \rightarrow a_1 \rightarrow a_1 \rightarrow a_2$$

does not define four times (0–3). There can only be three distinct times because the transition $a_1 \rightarrow a_1$ does not represent a change in state, and it is impossible for an observer of the system to distinguish the first and second occurrence of the state a_1 without reference to an external meta-system. Time is defined *relative* to state transition and not the other way around.

2.3 Data Abstraction and Re-timing

System states can be observed with respect to an abstraction mapping ψ . Such *data abstraction* occurs when the correctness of a system is defined with respect to a more abstract requirements specification. For example, a state a might represent the state of a microprocessor's *micro-architecture*. The state $\psi(a)$ then represents the state of the processor's *architecture*. That is, the state components that are of direct relevance to a machine or assembly code programmer. Through the process of data abstraction, a notion of temporal abstraction is induced. For example, if the mapping

$$a_0, a_5 \xrightarrow{\psi} b_0 \quad \text{and} \quad a_1, a_2, a_3, a_4 \xrightarrow{\psi} b_1$$

is applied to the state sequence

$$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_5 \rightarrow \dots$$

then the abstracted state sequence

$$b_0 \rightarrow b_1 \rightarrow b_1 \rightarrow b_1 \rightarrow b_1 \rightarrow b_0 \rightarrow \dots$$

has only two observable state changes, and would be perceived as the sequence

$$b_0 \rightarrow b_1 \rightarrow b_0 \rightarrow \dots$$

In this example temporal abstraction has occurred because six distinct times have been replaced by three times: see Figure 1.

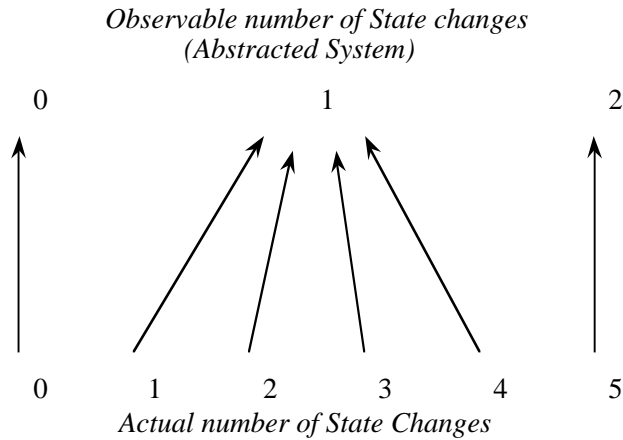


Figure 1: Time is redefined when, through a process of data abstraction, state transitions are no longer observable.

2.4 Related Work

Interesting work on pipelined microprocessors includes [42] on UINTA, a processor of moderate complexity, and its verification in HOL [16]; [29, 30] on AAMP5, a more complex processor, and its verification in PVS [32]; and [6] on a fragment of the DLX architecture [20]. More recently, superscalar processors have been addressed: in particular, the increased complexity of verification in the face of complex timing behaviour [41, 5, 38, 8, 29].

The intuitive models used by others in modelling and verifying [pipelined] microprocessors are conceptually similar to our own [18, 19, 12]. However, there are substantial differences, particularly in the approach to time, and timing abstraction. The main focus of related, formal work on microprocessors is the practical reality of developing techniques to successfully address more complex, and in some cases industrially-significant, examples (almost always in conjunction with, and tailored to, specific software tools). Our own work is concerned with developing a general formal framework for representing and verifying microprocessors within a uniform and well-developed algebraic theory. We have not, to date, applied software tools to significant examples.

In [42] systems are modelled as *state streams*: functions from time to state. Temporal and data abstraction functions are used to map between time and state at different levels of abstraction. In earlier work [40] (which also aims at a more general foundational framework), data and timing abstraction functions are separated (as in this paper). However, in [42], and related work on pipelined systems, data and timing abstraction are combined. This is because the view is taken that the values of specification state components are distributed in time at the level of abstraction of the

implementation. For example, the value of a data register reg in an implementation may correspond with a specification state at time t , and the value of the program counter pc with a specification state $t + n$, where n pipeline stages are required for an instruction to progress from initiation to completion. In this paper, we take the view that, rather than being temporally shifted, such state components are fundamentally different at the levels of specification and implementation (Sect. 4.3). Consequently, we maintain a separation between data and temporal abstraction functions.

The work of [29, 30] derives from [2, 34] on a simpler pipelined processor. In [2, 34], specification and implementation are modelled as state sequences, but time is not explicitly present; to synchronise the specification and implementation state sequences, multiple copies of specification states are inserted. In [29, 30], a different approach is taken. A *visible state* predicate is introduced which identifies those implementation states that should correspond to a specification state. This approach is modified, in a manner similar to that of [42], to cope with pipelining by distributing data in time. Again, in [29, 30], time is not explicitly present. A recent account of this work is [9].

It is not clear that temporal distribution of data (or combination of data and temporal abstraction) is required in practice. Substantial case studies to date (including a superscalar microprocessor in [11, 14]) have not required such techniques. Moreover, it may introduce extra complexities. For example, consider the above example of a user register reg in an implementation being mapped to a specification state at time t , and the corresponding program counter pc value being mapped to a specification state at time $t + n$. The value of pc relates to an instruction *entering* the pipeline, and reg to an instruction *leaving* the pipeline. Consequently, it is necessary to consider the entire time span of an instruction's execution cycle, which is likely, in a pipelined system, to overlap with other instructions. Our own approach only requires times at which instructions *complete* to be considered. In a pipelined system (though not a superscalar one), only one instruction will complete at a time.

In [6] a simple three-stage ALU pipeline, and a fragment of DLX are considered. Given a state Q or a pipelined implementation, a new state Q' is generated after executing one step of an instruction I . Both Q and Q' are then *flushed* by repeatedly *stalling* further execution (effectively filling them with no-ops). This results in two new states Q_f representing the (flushed) pipeline and Q'_f representing the (flushed) pipeline after executing instruction I . Q_f and Q'_f can be compared with appropriate specification states by simply projecting out the specification state elements. Note that there is no timing abstraction in this model: specification and implementation are both considered to take a single cycle to execute an instruction. This method is only applicable if some mechanism for stalling the pipeline is available. This is generally the case in real processors, though not in our

case study (Sect. 4).

Also of interest is [28] which again has a somewhat similar model of time. An injective, monotonic function f_P maps *abstract* time to *concrete* time, and is defined in terms of a predicate P . If $P(t_c)$ for some concrete time t_c , then there is an abstract time t_a such that $f_P(t_a) = t_c$. Predicate P is required to be true at an infinite number of times. The map f_P is similar to the *immersion* of Sect. 3.4.

Interesting earlier work on microprocessors includes the following. *Gordon's Computer* [15], a significant example since considered by others in [24, 35, 19]. *Viper* [7], which was partially verified in HOL. Landin's SECD machine [25] has been considered in [17, 3]. [23, 21, 22, 4] discuss a PDP-11-based processor and a more advanced successor. [1, 33] discuss parts of the Inmos T800 and T9000 Transputers, using an Occam-based transformation system.

3 Algebraic Formalisms

Computer systems are modelled in an algebraic framework using primitive recursive functions. We omit detailed discussion of algebraic specification methods here, and refer the reader to [10, 43, 27, 39]. Functions are defined, with equations, 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. For example,

$$(A_1, A_2, \dots, A_l \mid f_1, f_2, \dots, f_m)$$

denotes a many-sorted algebra 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 $1 \leq j \leq n$. The value n is called the *arity* of function f_i and if $n = 0$ then f_i is called a *constant*.

3.1 Clocks and Iterated Maps

This section formally defines *clocks* and *iterated map state functions*. A system starts at time zero in an *initial* state $h(a)$ where h is called an *initialisation function*. Subsequent system states are determined by a *next-state function* f and enumerated by a clock T . Clock enumeration is defined using a clock cycle successor function $+1$. The function h constrains the number of possible state sequences, thus enabling a more flexible definition of correctness. It is feasible for a system to be correct provided it starts in a

valid state. For example, most pipelined designs will not function correctly if the pipeline is initially filled with arbitrary data: see Sect. 4.

Definition 3.1 *A clock is an algebra $(T \mid 0, +1)$, where T is a set of clock cycles, $0 \in T$ is the initial clock cycle and $+1 : T \rightarrow T$ is the next (or successor) clock cycle function.*

A clock cycle need not represent a constant subdivision of time, but will denote an interval between significant events. The definition of ‘significant’ will depend on the level of temporal abstraction we are considering. For example, we might use an *instruction clock* to represent the execution of instructions in a microprocessor. Each cycle of the clock would typically last different amounts of real time, because instruction execution times vary in most processor implementations.

Definition 3.2 *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 the equations*

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

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

Typically, in a microprocessor, A will be a cartesian product of components representing registers and memories. By requiring f and h to be primitive recursive functions, or to have primitive recursive bounds, we eliminate all potential difficulties with partial functions, and non-termination. In practice, this condition is not restrictive.

Corollary 3.1 *If $F : T \times A \rightarrow A$ is an iterated map with initialisation function $h : A \rightarrow A$ and next-state function $f : A \rightarrow A$, then*

$$F(t, a) = (h \circ f^t)(a) = f^t(h(a)).$$

Note: The function composition $f \circ g$ is defined by $f \circ g(a) = g(f(a))$.

Definition 3.3 *A non-initialised iterated map is any iterated map F with initialisation function id_A , where $id_A : A \rightarrow A$ is the identity function defined by the equation $id_A(a) = a$. All iterated maps that are not non-initialised are called initialised iterated maps.*

3.2 Structural Abstraction: A Hierarchy of Algebras

To provide a degree of structural abstraction, system specifications are stratified into a number of distinct and well-defined levels. These levels constitute a hierarchy of algebras as follows.

1. The state algebra $(T, A \mid F)$ specifies the overall behaviour of a system, where A is a state-space, T is a clock carrier set and $F : T \times A \rightarrow A$ is a state function.
2. The next-state algebra $(T, A \mid 0, +1, h, f)$ specifies initialisation, state change and clock cycle enumeration, where $h : A \rightarrow A$ and $f : A \rightarrow A$ are initialisation and next-state functions respectively.
3. In the context of hardware specifications *machine algebras* contain the fundamental data and control operations of a device which are, in the case of digital hardware, typically operations ranging over bit vectors.

These three levels are all defined using a *fundamental algebra* which contains underlying function building operations: for example, definition by cases and primitive recursion.

3.3 Data Abstraction

Data abstraction is modelled as a surjective mapping between two state-spaces. Let $\psi : B \rightarrow A$ be a surjective map between two non-empty sets A and B . Surjectivity ensures that all abstract states in the set A have at least one representative in the set B . If all the states in A have exactly one representative in B then the map ψ is bijective and the state-spaces A and B are said to be at the same level of abstraction. Data abstraction maps are often 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 $n \leq m$. In this case, the implementation state-space B contains, without modification, all the components of the abstract state-space A (the sets B_{i_j}) together with components strictly unique to the implementation.

3.4 Temporal Abstraction: Retimings and Immersions

This section formally defines *retimings* (not to be confused with the retimings of [26]) and *immersions*. Two clocks are related using a temporal abstraction map, or retiming. Retimings are characterised by three properties: (i) cycle zero of one clock is always mapped to cycle zero of the other;

(ii) the mapping is surjective; and (iii) the mapping is monotonic. 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.

Definition 3.4 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 $Ret(S, T)$.

Definition 3.5 The immersion $\bar{\lambda}$ of a retiming $\lambda \in Ret(S, T)$ is defined by the equation

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

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

Note: Although an immersion is defined by unbounded minimalisation it is total because retimings are surjective. Given a retiming λ there exists a constant l_λ , such that, $l(\lambda)(t) < l_\lambda$ for all $t \in T$. Therefore, the minimalisation in Definition 3.5 can always be bounded by the constant l_λ .

Corollary 3.2 If $\lambda \in Ret(S, T)$ is a retiming then $(\bar{\lambda} \circ \lambda) = id_T$.

Definition 3.6 The function $start : Ret(S, T) \rightarrow [S \rightarrow S]$ is defined by the equation

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

Definition 3.7 The length function $l : Ret(S, T) \rightarrow [T \rightarrow S^+]$ is defined by the equation

$$l(\lambda)(t) = \bar{\lambda}(t + 1) - \bar{\lambda}(t).$$

Definitions 3.5 and 3.6 are illustrated with an examples in Figure 2, and by Definition 3.7 the following equivalences hold for the illustrated retiming: $l(\lambda)(0) = 4$ and $l(\lambda)(1) = 2$.

Definition 3.8 A clock S is faster than clock T (and clock T is slower than clock S) if there is a retiming λ from S to T . A clock S is strictly faster than clock T if S is faster than T with retiming λ and there exists a time $t \in T$ such that $l(\lambda)(t) > 1$. Clocks S and T are at the same speed if S is faster than T but not strictly faster.

3.5 State-Dependent and Uniform Retimings

Retimings, and consequently immersions, should be defined relative to state transition. If state evolution is deterministic then one state is sufficient to define an entire temporal abstraction map.

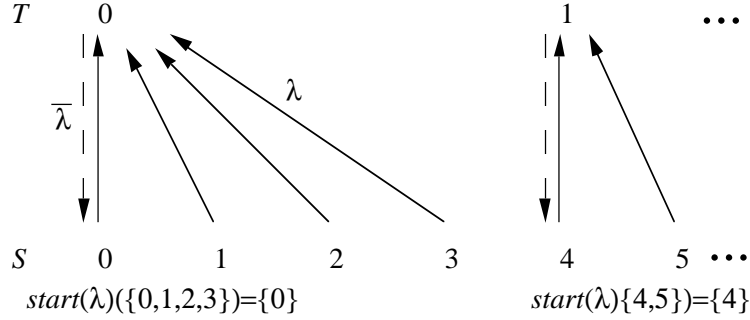


Figure 2: A retiming λ with associated immersion $\bar{\lambda}$.

Definition 3.9 A state-dependent retiming $\lambda : A \rightarrow \text{Ret}(S, T)$ is a map from states to retimings. The set of all state-dependent retimings from state-space A to retimings in $\text{Ret}(S, T)$ is denoted by $\text{Ret}(A, S, T)$.

Definition 3.10 The immersion $\bar{\lambda}$ of a state-dependent retiming λ is defined by the equation

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

The set of all immersions of state-dependent retimings in $\text{Ret}(A, S, T)$ is denoted by $\text{Imm}(A, S, T)$.

For each state of an implementation there is an associated state-dependent retiming. *Uniform* retimings provide a strong connection between the states generated by a state function F , from an initial state a , and the one retiming associated with the state a . This connection is achieved by associating a *duration* with each state in the state-space of F .

Definition 3.11 Let $F : S \times A \rightarrow A$ be an iterated map and $\text{dur} : A \rightarrow S^+$ 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 $\lambda \in \text{Ret}(A, S, T)$ such that, for all $a \in A$ and $t \in T$

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

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

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 $t \in T$ and $s \in S$.

Definition 3.12 *A state-dependent retiming $\lambda \in Ret(A, S, T)$ is uniform with respect to an iterated map $F : S \times A \rightarrow A$ if, and only if, there exists a function $dur : A \rightarrow S^+$ such that $\lambda \in URet_F^{dur}(A, S, T)$. The set of all uniform retimings with respect to F is denoted by $URet_F(A, S, T)$.*

In Sect. 2.3 it was shown that temporal abstraction is strongly related to data abstraction. Given a data abstraction $\psi : B \rightarrow A$ a simple definition for the duration map of a uniform retiming $\lambda \in URet_F(A, S, T)$ is

$$dur(a) = \text{least } s \in S^+ \text{ such that } \psi(a) \neq \psi(f^s(a)) \quad (1)$$

where f is the next-state function of F (definition 3.2). For simple case studies the duration map can be effectively defined in a constructive manner. That is, each duration is worked out in advance. However, this becomes increasingly difficult with complex examples [11, 14, 13].

Equation 1 provides a natural basis for the definition of a retiming based on the principle of data abstraction masking state change, and consequently inducing temporal abstraction. In Sect. 3.6 we show that temporal abstractions also have the dual rôle of establishing *when* an implementation is *correct*. In this rôle a retiming may diverge from the simple definition in equation 1 in order to prevent the observation of incorrect intermediate implementation states: in real examples, the elements of $\psi(a)$ can change value at different times. For example, in a simple microprogrammed processor, a *program counter* may be incremented at an early stage of an instruction cycle, and a result written to a user register near the end. A retiming can act as a temporal *filter*, selecting times of importance.

3.6 Implementation Correctness

This section provides an equational definition of correctness through the comparison of two state algebras: the state function of an implementation is compared with that of an abstract requirements specification. For this comparison, the state sequences specified by the implementation are mapped into the abstract [requirements] domain by the suitable application of a data abstraction map ψ and a temporal abstraction map λ . Data and temporal abstractions specify exactly *how* an implementation is correct and should be viewed as intrinsic parts of the design.

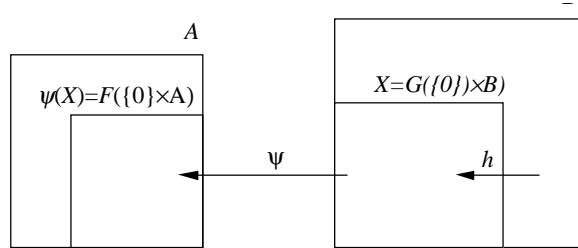


Figure 3: Completeness is preserved only if the *initial* implementation state-space X contains a representative for each *initial* specification state $\psi(X)$.

Definition 3.13 A state function $G : S \times B \rightarrow B$ can be called a correct implementation of a state function $F : T \times A \rightarrow A$ 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 $s = \text{start}(\lambda(b))(s)$

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

Corollary 3.3 A map G is a correct implementation of F with respect to λ and ψ if, and only if, the following diagram commutes for all $b \in B$ and $s = \text{start}(\lambda(b))(s)$

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

where $\hat{\lambda} : S \times B \rightarrow T$ is a cartesian form of $\lambda \in \text{Ret}(A, S, T)$ defined by the equation $\hat{\lambda}(s, b) = \lambda(b)(s)$ and $\hat{\psi} : S \times B \rightarrow B$ is defined by $\hat{\psi}(s, b) = \psi(b)$.

Correctness is required to hold at all ‘start’ clock cycles. That is, states of the implementation corresponding with [observable] specification states. This is expressed with the equation $s = \text{start}(\lambda(b))(s)$. All cycles such that $s = \text{start}(\lambda(b))(s)$ are enumerated by the immersion $\bar{\lambda} \in \text{Imm}(B, S, T)$.

The stipulation that the data abstraction map ψ is surjective means that all abstract states are representable at the implementation level, but this does not imply representatives need ever occur in the range of $G : S \times B \rightarrow B$. If the function G has an initialisation function $h : B \rightarrow B$ then the functionality of the implementation can be restricted in such a way that not all abstract state sequences can be generated. The following definition ensures that all valid initial abstract states can be represented by an initial implementation state (see Figure 3).

Definition 3.14 A map $G : S \times B \rightarrow B$ is a complete correct implementation of a map $F : T \times A \rightarrow A$ if, and only if, G is a correct implementation of F and

$$\psi(G(\{0\} \times B)) = F(\{0\} \times A).$$

Note: The set $H(\{0\} \times C)$ denotes the *image* of the set $\{0\} \times C$ under H . That is, the set $\{H(0, c) \mid c \in C\}$.

3.7 Time-Consistency and the One-Step Theorems

Iterated map state functions are *time-consistent* if they facilitate a process of staggered state evolution. The following question is addressed: for all times $s \in \bar{\lambda}(B)$, if the clock is reset to zero and the current state becomes an initial state, then is there any noticeable effect upon future state evolution? An iterated map $F : S \times A \rightarrow A$ is time-consistent if its initialisation function $h : A \rightarrow A$ characterises a state invariant. Expressed formally: $h(a) = a$ for all states $a \in F(\bar{\lambda}(A) \times A)$ in the range of F .

Definition 3.15 *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 $s_1 = \bar{\lambda}(F(s_2, a))(t_1)$.

Corollary 3.4 *If F is an iterated map with initialisation function h and next-state function f then F is time-consistent with respect to λ if, and only if, the following diagram commutes for all $a \in A$, $s_2 = \bar{\lambda}(a)(t_2)$ and $s_1 = \bar{\lambda}(F(s_2, a))(t_1)$*

$$\begin{array}{ccc} a & \xrightarrow{h \circ f^{s_2}} & a' = F(s_2, a) \\ \downarrow h \circ f^{s_1 + s_2} & & \downarrow h \circ f^{s_1} \\ F(s_1 + s_2, a) & \equiv & F(s_1, a') \end{array}$$

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

The following two results are called one-step theorems. Theorem 3.1 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 $t = 0, 1$.

Time-consistent iterated maps have the property that all possible occurrent states arise at time zero. Theorem 3.2 states that retiming uniformity and implementation time-consistency are sufficient conditions to enable correctness to be wholly verified by examining the two times $t = 0, 1$.

Theorem 3.1 *If $F : S \times A \rightarrow A$ is an iterated map with initialisation function $h : A \rightarrow A$ and if $\lambda \in \text{URet}_F^{\text{dur}}(A, S, T)$ is a uniform retiming then F is time-consistent with respect to λ if, and only if, for all $a \in A$*

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

Proof

See [11, 13].

Theorem 3.2 *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*

1. F is non-initialised, and
2. G is time-consistent with respect to λ

then G is a correct implementation of F if, and only if

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

Proof

See [11, 13].

Theorems 3.1 and 3.2 allow us to verify the correctness of systems represented as iterated maps without recourse to recursion.

4 An Abstract Pipeline Case Study

Two abstract pipelined designs, called P_1 and P_2 , are presented and completely verified, by hand, using term-rewriting under the correctness criteria of the one-step theorems. The intent of this work is to provide an elegant treatment of the subtle data and temporal abstraction aspects of pipelined organisations. A very simple non-pipelined requirements specification, called TR , is used. The device TR contains sufficient functionality to demonstrate the underlying temporal and invariance principles of pipelined designs. The device contains the following key components:

Memory: Temporal abstraction is especially significant when a computing device contains memory. Almost by definition, read-write memory keeps a history of past events (state changes). The device TR has two memories: one contains source data src , and the other contains computation results dst . The memories are addressed by registers: the memory source register msr , and the memory destination register mdr .

A Composite Operation. The device TR reads data from source memory to destination memory, and in the process the data is *transformed* using a composite operation $f = (f_1 \circ f_2 \circ \dots \circ f_n)$. The pipelined implementations perform the operations f_i *in time*.

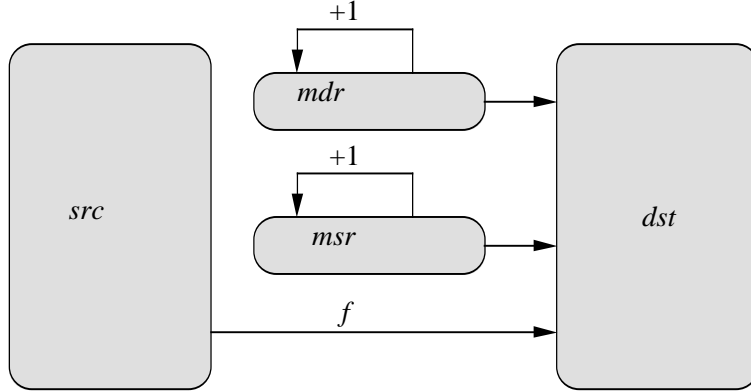


Figure 4: The abstract machine TR .

The implementations P_1 and P_2 exclude the temporal consequences of [instruction] dependencies. The temporal behaviour of real pipelined [processor] designs is heavily influenced by the instructions under execution: see [11, 14]. Two memories are used to simplify the example, by ensuring the contents of the pipeline are not rendered obsolete by the storage of data to the source memory.

4.1 Addressable Memory

Memory is modelled as a map from a *memory-address-space* to *memory words*. If MAR denotes a set of memory addresses and W denotes memory words, then the memory state-space is denoted $[MAR \rightarrow W]$. The memory algebra

$$([MAR \rightarrow W], MAR, W \mid \cdot(\cdot), \cdot[\cdot/\cdot])$$

contains two operations: a memory read function

$$\cdot(\cdot) : [MAR \rightarrow W] \times MAR \rightarrow W$$

and a memory substitution (write) function

$$\cdot[\cdot/\cdot] : [MAR \rightarrow W] \times MAR \times W \rightarrow [MAR \rightarrow W].$$

The memory word at address $a \in MAR$ of memory $m \in [MAR \rightarrow W]$ is denoted $m(a)$; and if the value $w \in W$ is stored at address a then the resultant memory is denoted $m[w/a]$. The memory substitution function is related to the memory read operation by the following equation

$$m[w/a](b) = \begin{cases} m(b), & \text{if } b \neq a; \\ w, & \text{if } b = a. \end{cases}$$

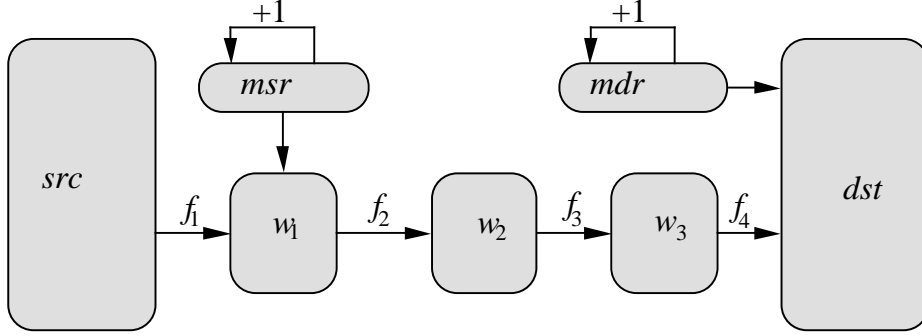


Figure 5: P_1 : A pipelined implementation of TR .

4.2 The Requirements Specification

The abstract device TR contains two memories and two memory-address registers. The memory state-space is $M = [MAR \rightarrow W]$ where W is any non-empty set, and the memory-address register state-space is MAR . The state-space of TR is

$$State_{TR} = M \times MAR \times MAR \times M.$$

The device TR is specified with state function $TR : T \times State_{TR} \rightarrow State_{TR}$ and next-state function $tr : State_{TR} \rightarrow State_{TR}$

$$\begin{aligned} TR(0, dst, msr, mdr, src) &= (src, msr, mdr, dst), \\ TR(t+1, src, msr, mdr, dst) &= tr(TR(t, src, msr, mdr, dst)), \\ tr(src, msr, mdr, dst) &= (src, msr + 1, mdr + 1, \\ &\quad dst[f(src(msr))/mdr]) \end{aligned}$$

where $src \in M$, $msr \in MAR$, $mdr \in MAR$ and $dst \in M$. The next-state function tr updates the destination memory dst at location mdr with $f(src(msr))$, and increments both memory-address registers. The primitive recursive function $f : W \rightarrow W$ is not explicitly defined, but the two pipelined implementations assume $f = (f_1 \circ f_2 \circ f_3 \circ f_4)$ for some

$$f_1 : W \rightarrow W_1, \quad f_2 : W_1 \rightarrow W_2, \quad f_3 : W_2 \rightarrow W_3, \quad f_4 : W_3 \rightarrow W$$

where W_1 , W_2 and W_3 are arbitrary non-empty sets. For brevity, $f_1 \circ f_2 : W \rightarrow W_2$ is denoted \vec{f}_2 , and $\vec{f}_2 \circ f_3 : W \rightarrow W_3$ is denoted \vec{f}_3 .

4.3 A Permanently Full Four-Stage Pipeline

A four-stage pipelined implementation of TR , called P_1 , is shown in Figure 5. Three additional state components w_1 , w_2 and w_3 form a pipeline by storing

intermediate computations of the operation $f = (f_1 \circ f_2 \circ f_3 \circ f_4)$. The state-space of P_1 is

$$State_{P_1} = M \times MAR \times W_1 \times W_2 \times W_3 \times MAR \times M.$$

The iterated map state function $P_1 : S \times State_{P_1} \rightarrow State_{P_1}$ is defined by the equations

$$\begin{aligned} P_1(0, \vec{\sigma}) &= p_1^0(\vec{\sigma}), \\ P_1(s+1, \vec{\sigma}) &= p_1(P_1(s, \vec{\sigma})) \end{aligned}$$

where $\vec{\sigma} = (src, msr, w_1, w_2, w_3, mdr, dst)$,

$$\begin{aligned} p_1^0(\vec{\sigma}) &= (src, msr, f_1(src(msr-1)), \\ &\vec{f}_2(src(msr-2)), \vec{f}_3(src(msr-3)), mdr, dst) \end{aligned}$$

and

$$\begin{aligned} p_1(\vec{\sigma}) &= (src, msr+1, f_1(src(msr)), \\ &f_2(w_1), f_3(w_2), mdr+1, dst[f_4(w_3)/mdr]). \end{aligned}$$

The initialisation function $p_1^0 : State_{P_1} \rightarrow State_{P_1}$ establishes a full pipeline by ensuring the state of component w_i corresponds with source data from memory-cell address $msr - i$, after the appropriate incremental application of the operations f_j , for all $j \leq i$. For example, at the third pipeline stage $w_3 = \vec{f}_3(msr-3) = f_1 \circ f_2 \circ f_3(msr-3)$. Initialisation function p_1^0 is as weak as possible: if $\vec{\sigma}$ is already consistent with correct future execution (that is, the pipeline is already correctly initialised), then $\vec{\sigma} = p_1^0(\vec{\sigma})$.

The next-state function $p_1 : State_{P_1} \rightarrow State_{P_1}$ maintains a full pipeline by forwarding computation results along the pipeline. For example, w_3 stores $f_3(w_2)$. The last stage of the pipeline applies the operation f_4 to component w_3 and stores the result in the memory dst at the address mdr .

The temporal relationship between P_1 and TR is trivial: a memory substitution occurs upon every cycle of the clock T and on every cycle of the clock S , therefore P_1 and TR are at the same level of temporal abstraction. That is, they are related by the retiming $\lambda \in Ret(State_{P_1}, S, T)$ with $\lambda(a)(s) = s$. This may seem counter-intuitive because the main purpose of pipelining is to increase temporal performance. One must remember that the clocks T and S enumerate state change and do not directly represent temporal performance in the physical sense. Given a clock $Time$ enumerating the state change of a physical clock with precision $1\mu s$ then the retimings $\lambda_1 \in Ret(Time, S)$ and $\lambda_2 \in Ret(Time, T)$ express the temporal characteristics of P_1 and TR respectively. For example, if each cycle of clock S lasts $1\mu s$, then it is reasonable to assume each cycle of clock T lasts $4\mu s$, because each operation f_i should be a [more trivial] stage in the computation of the

complex operation f . This is illustrated in Figure 6 where the retiming λ is plotted with respect to the clock $Time$.

Care must be taken when defining data abstraction $\psi : State_{P_1} \rightarrow State_{TR}$. A naïve first attempt to define ψ is the projection

$$\psi(src, msr, w_1, w_2, w_3, mdr, dst) = (src, msr, mdr, dst).$$

This definition is flawed by misconstruing the memory source register msr to be the same component at both levels of abstraction. Observe that, by the definition of p_1 upon all cycles of clock $S+ = S - \{0\}$ the memory substitution

$$dst[f(src(msr - 3))/mdr]$$

is performed, and on all cycles of clock T^+ the substitution

$$dst[f(src(msr))/mdr]$$

occurs. This is incompatible with the presumption that the msr of the specification and implementation are *directly* equivalent.

One method of perceiving the relationship between the memory source registers is that the implementation msr is a temporally advanced version of the specification msr . In our view, this is misleading, and suggests the components are still one in the same. Instead we regard the components as fundamentally different, because our data abstraction is temporally invariant. The rôle of msr is dictated by whether the pipeline is full or empty, and this is [primarily] a property of state not time. For any fixed time, the specification msr represents the location of source data, but at the implementation level msr is a component used to fill the pipeline and $msr - 3$ is the required location of source data. That is, a *function of the current* value of msr (and possibly other state components); rather than the value of msr from some other time. By taking this view, we are able to maintain the division between data and temporal abstraction functions, which are more conveniently defined separately.

Proposition 4.1 *The map P_1 is a correct implementation of TR with respect to data abstraction map $\psi : State_{P_1} \rightarrow State_{TR}$*

$$\psi(src, msr, w_1, w_2, w_3, mdr, dst) = (src, msr - 3, mdr, dst)$$

and uniform retiming $\lambda \in URet_{P_1}^{dur}(State_{P_1}, S, T)$, where duration function $dur : State_{P_1} \rightarrow S^+$ is defined by the equation

$$dur(src, msr, w_1, w_2, w_3, mdr, dst) = 1.$$

Proof

Banished to the Appendix.

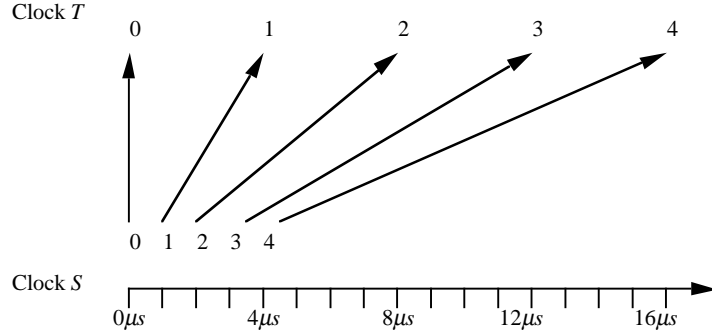


Figure 6: The temporal behaviour of the first implementation. There is a one-to-one correspondence between the clocks S and T , but when related to a physical measure of time, a performance increase is observed for the pipelined implementation P_1 .

4.4 A Self-Initialising Four-Stage Pipeline

The implementation of Sect. 4.3 assumes that the pipeline is permanently full, but in practice pipelines exhibit different stages of operation. For example, a pipeline may be flushed or emptied. Pipelined microprocessors must flush the instruction pipeline when conditional-branch prediction fails and, for example, the pipeline contains instructions at the addresses $pc+1, pc+2, \dots$ but should contain the instructions at the addresses $dst, dst+1, \dots$ where dst is the branch destination. We fill the pipeline using a counter $ctr \in \{1, 2, 3, 4\}$. If $ctr = 1$ then the pipeline is assumed to be full and the implementation maintains the functionality of the first implementation. If $ctr = 4$ then the pipeline is assumed to be empty with the pipeline components w_1, w_2 and w_3 containing *junk* values. The pipeline is filled by decrementing ctr while ensuring junk values are not stored in the destination memory dst until $ctr = 1$.

The state-space $State_{P_2}$ is an expansion of $State_{P_1}$ to include the counter

$$State_{P_2} = \{1, 2, 3, 4\} \times State_{P_1}$$

The iterated map state function $P_2 : S \times State_{P_2} \rightarrow State_{P_2}$ is defined by the equations

$$\begin{aligned} P_2(0, \vec{\sigma}) &= p_2^0(\vec{\sigma}), \\ P_2(s+1, \vec{\sigma}) &= p_2(P_2(s, \vec{\sigma})) \end{aligned}$$

where $\vec{\sigma} = (ctr, src, msr, w_1, w_2, w_3, mdr, dst)$,

$$p_2^0(\vec{\sigma}) = \begin{cases} (4, src, msr, w_1, w_2, w_3, mdr, dst), & \text{if } ctr > 1; \\ (1, p_1^0(src, msr, w_1, w_2, w_3, mdr, dst)), & \text{if } ctr = 1 \end{cases}$$

and

$$p_2(\vec{\sigma}) = \begin{cases} (ctr - 1, src, msr + 1, f_1(src(msr))), \\ \quad f_2(w_1), f_3(w_2), mdr, dst), & \text{if } ctr > 1; \\ (1, p_1(src, msr, w_1, w_2, w_3, mdr, dst)), & \text{if } ctr = 1. \end{cases}$$

The initialisation function $p_2^0 : State_{P_2} \rightarrow State_{P_2}$ ensures the device is either empty or full at cycle zero. In the case that it is empty, we need take no steps to ensure that the intermediate state components w_1, w_2, w_3 contain values consistent with correct future execution; if it full, then all of w_1, w_2, w_3 must be consistently initialised. We choose to map the intermediate pipeline stages when $ctr = 2$ or $ctr = 3$ to an empty pipeline by setting $ctr = 4$. This is a stronger initialisation function than is strictly necessary: we could choose to define functions to consistently initialise a partly-filled pipeline. However, in this particular example, the simpler definition is adequate, and does not violate time-consistency because, like the first implementation, once the pipeline is full it remains so, and therefore these intermediate states will never re-occur. If it was possible in our example for the pipeline to become emptied during operation, it would be necessary to consider these cases.

If $ctr = 1$ then, for the the state of the pipeline to be consistent with correct execution, w_1, w_2 and w_3 should contain $f_1(src(msr - 1))$, $\vec{f}_2(src(msr - 2))$ and $\vec{f}_3(src(msr - 3))$. Hence we use p_1^0 to initialise the pipeline, since if initial state $\vec{\sigma}$ is already correctly initialised, $\vec{\sigma} = p_1^0(\vec{\sigma})$.

The next-state function $p_2 : State_{P_2} \rightarrow State_{P_2}$ provides for two cases: if $ctr = 1$ then the pipeline is full and p_2 is [nearly] identical to the next-state function p_1 . But if $ctr > 1$ then the pipeline is progressively filled. This means ctr is decremented while the dst memory is unaltered to prevent storing junk values.

The second implementation P_2 is a correct implementation of TR . The temporal relationship between P_2 and TR is more in-line with the classical model of pipelines. The duration associated with any given state is directly proportional to the value of the counter ctr . That is, the extent to which the pipeline is full. By the construction of p_2 , this gives two cases: four cycles when the pipeline is empty and one cycle when the pipeline is full. The retiming $\lambda \in Ret(State_{P_2}, S, T)$ is illustrated in Figure 7 for the case of an empty initial pipeline state. The first four cycles are used to fill the pipeline and after that Figure 7 is identical to Figure 6 with an offset of $4\mu s$. Note: The speed of the pipeline is dictated by the slowest pipeline operation and in practice the cycles $s = 4$ and $t = 1$ would not correspond with the same physical time. In practice, the four pipeline stages would take longer than a single application of the operation f . The performance increase of a pipelined design comes from maintaining a full pipeline: one would expect $t/4 < t' < t$, where t' is the duration of the slowest pipeline stage and t is the duration of a monolithic non-pipelined implementation.

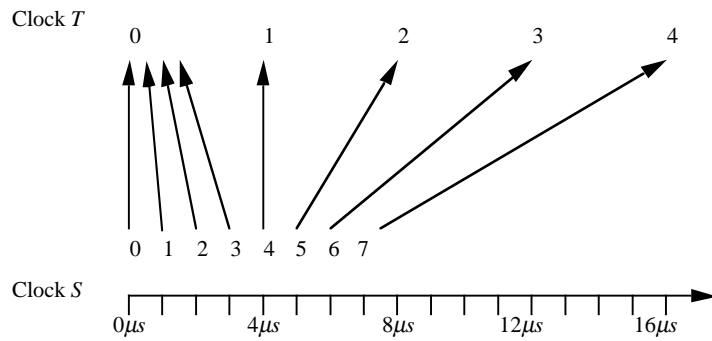


Figure 7: The temporal behaviour of the second implementation. The pipelined is initially empty. A one-to-one correspondence between the clocks S and T is established after four implementation cycles when the pipeline becomes full.

The rôle of the memory source register is dependent on whether the pipeline is full or empty. If the pipeline is full then the data abstraction $\psi : State_{P_2} \rightarrow State_{TR}$ is [nearly] identical to that of Proposition 4.1. If the pipeline is empty, then the memory source register of implementation P_2 is identical to the specification's memory source register. The rôle of the m_{sr} in the implementation is *indirectly* related to time (because the pipeline becomes full in time), but its relationship to m_{sr} in the specification is defined by the value of ctr .

Proposition 4.2 *The map P_2 is a correct implementation of TR with respect to data abstraction map $\psi : State_{P_2} \rightarrow State_{TR}$*

$$\psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = \begin{cases} (src, msr, mdr, dst), & \text{if } ctr > 1; \\ (src, msr - 3, mdr, dst), & \text{if } ctr = 1, \end{cases}$$

and uniform retiming $\lambda \in URet_{P_2}^{dur}(State_{P_2}, S, T)$ where duration function $dur : State_{P_2} \rightarrow S^+$ is defined by the equation

$$dur(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = ctr.$$

Proof

Banished to the Appendix.

5 Initialisation and Abstraction

The purpose of the abstract case studies in Sect. 4.3 and Sect. 4.4 is to emphasise the distinct rôles of initialisation, data abstraction and temporal abstraction. Initialisation functions identify the desired behaviour of an implementation and directly affect verification efforts. This can be seen by comparing the second implementation P_2 with the first P_1 . The second implementation P_2 is an implementation of P_1 with respect to the data abstraction $\psi : State_{P_2} \rightarrow State_{P_1}$ and the retiming λ , both defined in Proposition 4.2. There are two cases to consider in data abstraction map ψ : (i) if the pipeline of the second implementation P_2 is empty ($ctr > 1$) then the msr of P_1 is three addresses ahead because its pipeline is full; and (ii) if the second implementation P_2 is full, then the first and second implementations are identical once the counter ctr is hidden. The first implementation is more abstract because it does not specify how the pipeline is initially full — it just is. Nevertheless if the initialisation function for the second implementation was

$$p\eta(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = (1, src, msr, w_1, w_2, w_3, mdr, dst)$$

then the initial pipeline filling behaviour would never manifest itself and the two implementations are at the same level of abstraction. Therefore, when defining an initialisation function one must keep in mind two concerns: (i) it must be sufficiently weak to ensure the implementation is time-consistent and complete; and (ii) it can be even weaker to capture the desired initial implementation behaviour. There is little point in proving the second implementation is correct if the pipeline is always full. In the correctness

definitions of [29, 6] it is not possible to define such systems (that is, a pipeline which is only empty at cycle zero) because an explicit model of time is not used. Our correctness definition is more expressive and general by virtue of an explicit and well-developed temporal model.

Data abstraction has been treated exclusively as a function of state, but not necessarily as a pure projection. Case-based data abstraction is used when the implementation exhibits different stages of operation. For example, data abstraction for the second implementation P_2 is adapted to accommodate the implications of empty, together with full, pipelines. It is possible to construct examples where data abstraction *must be* temporally dependent but it is believed that such examples would be forced and not reflective of actual pipelined microprocessors. The purpose of microprocessors is to implement the program-level behaviour of an architecture. Therefore it is unrealistic that the abstract parts of the implementation state cannot be readily determined, at a fixed time, from a state corresponding with the completion of an instruction. Although data abstraction might not be a pure projection it should not be necessary to ‘look into the future’ when determining the specification state.

Temporal abstraction has been modelled in a clean manner using duration maps. The first implementation performs a destination memory substitution on every machine cycle. Therefore

$$dur(src, msr, w_1, w_2, w_3, mdr, dst) = 1.$$

The second implementation might be initially empty. Therefore

$$dur(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = ctr.$$

At cycle zero the component ctr either has the value four or one, therefore the definition above is effectively equivalent to

$$dur(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = \begin{cases} 1, & \text{if } ctr = 1; \\ 4, & \text{if } ctr = 2, 3, 4, \end{cases}$$

since initially the pipeline is forced to be either full or empty, and once full remains so.

6 Further Considerations

We have shown how the algebraic techniques developed by Harman and Tucker [18, 19] and extended by Fox and Harman [12, 11, 14] can be systematically applied to pipelined systems, by means of a simple example. These methods have been applied to substantially more complex examples: in [11], a pipelined implementation of an microprocessor is defined, and formally verified; and in [11, 14], a superscalar implementation of the same processor is defined.

All formal verifications to date, with the exception of some small experiments, have been performed manually. For this reason, the superscalar implementation in [11, 14] has not been verified, because it is too complex. Our main aim is a systematic formal framework for representing microprocessors and related systems, and not [mechanised] formal verification. However, we are interested in developing software support for our techniques: largely, because of the complexity of examples we wish to consider. We intend to adopt existing tools rather than developing our own. By establishing a general theoretical framework, we have ensured our techniques are suited to a range of software tools.

We have developed a systematic framework for defining data and temporal abstraction maps, and maintain a separation between the two. We have shown how formal verification can be simplified given easily-established conditions. It is possible to define temporal abstraction in a logical and clean manner because all our implementations and specifications are deterministic. Although this has not so far proved restrictive, we can consider circumstances where that may not be the case. For example, we have assumed in the case study in Sect. 4 that memory access takes one clock cycle. Given that our definition of a clock in Sect. 3.1 permits clock cycles to be variable in length, we can simply use a clock in which a new cycle only starts when all memories are ready. However, we may wish to formally examine the temporal behaviour of the memory system, possibly in conjunction with a cache. While we could do this deterministically, in practice it may become impractically complex. In addition, we may wish to consider *transactional* systems, in which memory access requests are made, and computation continues until the access is signalled to be complete. As currently formulated, our model does not lend itself to such systems. We are currently investigating the addition of extra levels of abstraction to address this. We are also attempting to unify our models of microprocessors with the closely-related models of K Stephenson for languages and compilers [36]: in particular, work on the *Java Virtual Machine* [37]. In this way, we aim to produce a uniform theoretical model that ranges from high-level programming languages to hardware, with well-defined abstraction maps between each level.

References

- [1] D May G Barrett and D Shepard. Designing chips that work. In C A R Hoare and M J C Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice-Hall, 1992.
- [2] M Bickford and M Srivas. Verification of a pipelined processor using Clio. In M Leeser and G Brown, editors, *Proceedings of the Mathematical Sciences Institute Workshop on Hardware Specification, Verification*

- and Synthesis: Mathematical Aspects*, pages 307 – 332. Lecture Notes in Computer Science 408, Springer-Verlag, 1990.
- [3] G Birtwistle and B Graham. Verifying SECD in HOL. In J Staunstrup, editor, *Formal Methods for VLSI Design*, pages 129 – 177. North-Holland, 1990.
 - [4] B Bose and S D Johnson. DDD-FM9001: Derivation of a verified microprocessor. In L Pierre G Milne, editor, *Correct Hardware Design and Verification Methods*, pages 191 – 202. Lecture Notes in Computer Science 683, Springer-Verlag, 1993.
 - [5] J Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference*, 1996.
 - [6] 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.
 - [7] A Cohn. A proof of correctness of the Viper microprocessor: the first levels. In G Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27 – 72. Kluwer Academic Publishers, 1987.
 - [8] 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.
 - [9] 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.
 - [10] H Ehrig and B Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. EATCS Monograph vol. 6, Springer-Verlag, 1985.
 - [11] A C J Fox. *Algebraic Representation of Advanced Microprocessors*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1998.
 - [12] 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.

- [13] A C J Fox and N A Harman. Algebraic models of correctness for microprocessors. Technical Report CSR 8-98 (submitted to *Formal Aspects of Computer Science*), University of Wales Swansea, 1998.
- [14] 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*. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
- [15] M J C Gordon. Proving a computer correct with the LCF-LSM hardware verification system. Technical Report 42, Computer Laboratory, University of Cambridge, 1983.
- [16] M J C Gordon and T F Melham. *Introduction to HOL*. Cambridge University Press, 1993.
- [17] B Graham. *The SECD Microprocessor: a Verification Case Study*. Kluwer, 1992.
- [18] N A Harman and J V Tucker. Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica*, 33:421 – 456, 1996.
- [19] 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.
- [20] J L Hennessy and D A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1996.
- [21] W Hunt. A formal HDL and its use in the FM9001 verification. In C A R Hoare and M Gordon, editors, *Mechanized Reasoning in Hardware Design*. Prentice-Hall, 1992.
- [22] W Hunt. *FM8501: A Verified Microprocessor*. Lecture Notes on Artificial Intelligence 795, Springer Verlag, 1994.
- [23] W A Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429 – 460, 1989.
- [24] J Joyce. Formal verification and implementation of a microprocessor. In G Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129 – 159. Kluwer Academic Publishers, 1987.
- [25] P Landin. On the mechanical evaluation of expressions. *Computer Journal*, 6:308 – 320, 1963.

- [26] 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.
- [27] 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.
- [28] T F Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press Tracts in Theoretical Computer Science 31, 1993.
- [29] S Miller and M Srivas. Formal verification of an avionics microprocessor. Technical report, SRI International Computer Science Laboratory CSL-95-04, 1995.
- [30] 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.
- [31] B Möller and J V Tucker. *Prospects for Hardware Foundations*. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
- [32] 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.
- [33] W Roscoe. Occam in the specification and verification of microprocessors. In C A R Hoare and M J C Gordon, editors, *Mechanized Reasoning and Hardware Design*. Prentice-Hall, 1992.
- [34] M Srivas and M Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52 – 64, 1991.
- [35] V Stavridou. *Formal Specification of Digital Systems*. Cambridge University Press Tracts in Theoretical Computer Science 37, 1993.
- [36] K Stephenson. *An Algebraic Approach to Syntax, Semantics and Compilation*. PhD thesis, University of Wales Swansea Computer Science Department, 1996.
- [37] 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.
- [38] J Su, D Dill, and C Barrett. Automatic generation of invariants in processor verification. In A Camilleri M Srivas, editor, *Formal Methods*

in *Computer-Aided Design*, pages 377 – 388. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.

- [39] W Wechler. *Universal Algebra for Computer Scientists*. EATCS Monograph, Springer-Verlag, 1991.
- [40] P Windley. A theory of generic interpreters. In L Pierre G Milne, editor, *Correct Hardware Design and Verification Methods*, pages 122 – 134. Lecture Notes in Computer Science 683, Springer-Verlag, 1993.
- [41] 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.
- [42] P Windley and M Coe. A correctness model for pipelined microprocessors. In *Proceedings of the 2nd Conference on Theorem Provers in Circuit Design*, 1994.
- [43] 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.

Appendix: Banished Proofs

Proof: Proposition 4.1 The map TR does not have an initialisation function and therefore Theorem 3.2 can be used if P_1 is time-consistent. This gives the following four proof obligations

$$\begin{aligned}
P_1(0, src, msr, w_1, w_2, w_3, mdr, dst) &= \\
& p_1^0(P_1(0, src, msr, w_1, w_2, w_3, mdr, dst)), \\
P_1(1, src, msr, w_1, w_2, w_3, mdr, dst) &= \\
& p_1^0(P_1(1, src, msr, w_1, w_2, w_3, mdr, dst)), \\
TR(0, \psi(src, msr, w_1, w_2, w_3, mdr, dst)) &= \\
& \psi(P_1(0, src, msr, w_1, w_2, w_3, mdr, dst)), \\
TR(1, \psi(src, msr, w_1, w_2, w_3, mdr, dst)) &= \\
& \psi(P_1(1, src, msr, w_1, w_2, w_3, mdr, dst)).
\end{aligned}$$

At time $t = 0$:

$$\begin{aligned}
& P_1(0, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_1^0(src, msr, w_1, w_2, w_3, mdr, dst) \\
&= (src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= p_1^0(src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= p_1^0(P_1(0, src, msr, w_1, w_2, w_3, mdr, dst)).
\end{aligned}$$

At time $t = 1$:

$$\begin{aligned}
& P_1(1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_1(P_1(0, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= p_1(src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= (src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \text{ because } f = (f_1 \circ f_2 \circ f_3 \circ f_4) \\
&= p_1^0(src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \\
&= p_1^0(P_1(1, src, msr, w_1, w_2, w_3, mdr, dst)).
\end{aligned}$$

At time $t = 0$:

$$\begin{aligned}
& TR(0, \psi(src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= TR(0, src, dst, msr - 3, mdr) \\
&= (src, dst, msr - 3, mdr) \\
&= \psi(src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= \psi(P_1(0, src, msr, w_1, w_2, w_3, mdr, dst)).
\end{aligned}$$

Finally, at time $t = 1$:

$$\begin{aligned}
& TR(1, \psi(src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= tr(TR(0, src, dst, msr - 3, mdr)) \\
&= tr(src, dst, msr - 3, mdr) \\
&= (src, dst[f(src(msr - 3))/mdr], msr - 2, mdr + 1) \\
&= \psi(src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \\
&= \psi(P_1(1, src, msr, w_1, w_2, w_3, mdr, dst)).
\end{aligned}$$

Proof: Proposition 4.2 The map P_2 is a correct implementation of TR if the following four equations hold with $\vec{\sigma} = (ctr, src, msr, w_1, w_2, w_3, mdr, dst) \in State_{P_2}$

$$\begin{aligned}
P_2(0, \vec{\sigma}) &= p_2^0(P_2(0, \vec{\sigma})), \\
P_2(\bar{\lambda}(\vec{\sigma})(1), \vec{\sigma}) &= p_2^0(P_2(\bar{\lambda}(\vec{\sigma})(1), \vec{\sigma})), \\
TR(0, \psi(\vec{\sigma})) &= \psi(P_2(0, \vec{\sigma})), \\
TR(1, \psi(\vec{\sigma})) &= \psi(P_2(\bar{\lambda}(\vec{\sigma})(1), \vec{\sigma})).
\end{aligned}$$

Each of these four conditions is split into two sub-cases: $ctr = 1$ and $ctr = 2, 3, 4$. This gives the eight cases below.

Case 1: $t = 0$ and $ctr = 1$

$$\begin{aligned}
& P_2(0, 1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2^0(1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= (1, src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= p_2^0(1, src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= p_2^0(P_2(0, 1, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 2: $t = 0$ and $ctr = 2, 3, 4$

$$\begin{aligned}
& P_2(0, ctr, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2^0(ctr, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= (4, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2^0(4, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2^0(P_2(0, ctr, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 3: $t = 1$ and $ctr = 1$

$$\begin{aligned}
& P_2(\bar{\lambda}(1, src, msr, w_1, w_2, w_3, mdr, dst)(1), 1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= P_2(1, 1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2(P_2(0, 1, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= p_2(1, src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= (1, src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \\
&= p_2^0(1, src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \\
&= p_2^0(P_2(\bar{\lambda}(1, src, msr, w_1, w_2, w_3, mdr, dst)(1), \\
&\quad 1, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 4: $t = 1$ and $ctr = 2, 3, 4$

$$\begin{aligned}
& P_2(\bar{\lambda}(ctr, src, msr, w_1, w_2, w_3, mdr, dst)(1), ctr, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= P_2(4, ctr, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= p_2^4(P_2(0, ctr, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= (1, src, msr + 4, f_1(src(msr + 3)), \vec{f}_2(src(msr + 2)), \vec{f}_3(src(msr + 1)), \\
&\quad mdr + 1, dst[f(src(msr))/mdr]) \\
&= p_2^0(1, src, msr + 4, f_1(src(msr + 3)), \vec{f}_2(src(msr + 2)), \vec{f}_3(src(msr + 1)), \\
&\quad mdr + 1, dst[f(src(msr))/mdr]) \\
&= p_2^0(P_2(\bar{\lambda}(ctr, src, msr, w_1, w_2, w_3, mdr, dst)(1), \\
&\quad ctr, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 5: $t = 0$ and $ctr = 1$

$$\begin{aligned}
& TR(0, \psi(1, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= \psi(1, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= (src, dst, msr - 3, mdr) \\
&= \psi(1, src, msr, f_1(src(msr - 1)), \vec{f}_2(src(msr - 2)), \\
&\quad \vec{f}_3(src(msr - 3)), mdr, dst) \\
&= \psi(P_2(0, 1, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 6: $t = 0$ and $ctr = 2, 3, 4$

$$\begin{aligned}
& TR(0, \psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= \psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst) \\
&= (src, dst, msr, mdr) \\
&= \psi(P_2(0, ctr, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 7: $t = 1$ and $ctr = 1$

$$\begin{aligned}
& TR(1, \psi(1, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= tr(\psi(1, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= (src, dst[f(src(msr))/mdr], msr - 2, mdr + 1) \\
&= \psi(1, src, msr + 1, f_1(src(msr)), \vec{f}_2(src(msr - 1)), \vec{f}_3(src(msr - 2)), \\
&\quad mdr + 1, dst[f(src(msr - 3))/mdr]) \\
&= \psi(P_2(\bar{\lambda}(1, src, msr, w_1, w_2, w_3, mdr, dst)(1), \\
&\quad 1, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$

Case 8: $t = 1$ and $ctr = 2, 3, 4$

$$\begin{aligned}
& TR(1, \psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= tr(\psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst)) \\
&= (src, dst[f(src(msr))/mdr], msr + 1, mdr + 1) \\
&= \psi(1, src, msr + 4, f_1(src(msr + 3)), \vec{f}_2(src(msr + 2)), \vec{f}_3(src(msr + 1)), \\
&\quad mdr + 1, dst[f(src(msr))/mdr]) \\
&= \psi(P_2(\bar{\lambda}(ctr, src, msr, w_1, w_2, w_3, mdr, dst)(1), \\
&\quad ctr, src, msr, w_1, w_2, w_3, mdr, dst))
\end{aligned}$$