

Modelling SMT and CMT Processors: A Simple Case Study

DRAFT

N A Harman

Department of Computer Science, University of Wales Swansea, Swansea SA2 8PP, UK
n.a.harman@swan.ac.uk

May 7, 2007

Abstract

This paper builds on a series examining models of pipelined and superscalar microprocessors and their correctness by extending them to Simultaneous Multithreaded (SMT) and Chip-Level Multithreaded (CMT) processors. SMT and CMT implementations behave, to the programmer, like separate processors (virtual in the case of SMT) that communicate by means of shared state. The timing relationships are complex: the multiple (virtual) processors effectively operate over separate clocks, that are related by their temporal relationship with the corresponding implementation and its state. This relationship is likely to be complex since SMT processors are inherently also superscalar. In practice, CMT processors are also likely to be superscalar.

The existing tools for modeling microprocessors and their correctness are extended to SMT and CMT systems. The model is designed to preserve the (likely large) investment in time and effort made in developing non-SMT/CMT processor models. The model is illustrated by a simple, dual-core pipelined processor example. The model also accurately reflects the inevitable presence of aspects of a SMT/CMT processor's implementation in the programmer-visible behaviour.

In response to difficulties in increasing instruction-level parallelism in pipelined and superscalar processor implementations, microprocessor manufacturers are implementing designs that behave like multiple programmer-level processors. Such implementations generally show performance gains which can be effectively exploited by appropriately-written program code. There are currently two main approaches to the implementation of such devices. *Chip-Level Multithreaded* (CMT), or *multi-core* processors (for example, Intel's *Core Duo*) duplicate the majority of the instruction pipeline with the exception of the cache (and memory, which though not part of the processor itself is commonly included in formal models). *Simultaneous Multi-Threaded* (SMT)¹ processors interleave multiple threads of execution within the same pipeline. In practice, to be effective, SMT pipelines must be superscalar, and substantial resources must be duplicated. However, by sharing as much of the pipeline as possible between threads of execution, hardware utilization can be increased resulting in performance gains.

This paper addresses the following technical questions: given some physical implementation I that presents itself as multiple [actual or virtual] microprocessor implementations I_1, \dots, I_n , how can I and its corresponding programmer-level abstractions S_1, \dots, S_n be modeled, where each S_i may communicate via *shared state*? And what does it mean for I to correctly implement S_1, \dots, S_n ? The model developed is illustrated using a dual core pipelined example, based on that in [4]. The focus here is the example and how the model is applied: for a more theoretical discussion, including proofs, see [11, 10]. The underlying model for this

¹Called Hyperthreading by Intel.

work is *universal algebra*, in particular *many-sorted initial algebra* [18]. However, no knowledge of universal algebra is required, and the techniques themselves map straightforwardly to other formalisms. For example, [5, 6] use the work in [9, 7] (on which this paper is based) to verify ARM6 using HOL.

1 Related Work

TO BE REVISED AND UPDATED.

There is a substantial body of work devoted to microprocessor verification and only a brief summary is possible here. A common characteristic of much of the work is the need to address a specific, usually complex² example. Neglecting early work from the 1980s³ and earlier, a landmark leading to much subsequent work is [2]. Key concepts like the relationship between time at different levels of abstraction, and how it can be addressed appear [14]. Work appears on pipelined and superscalar models with parallels with our own: substantial differences are that although the concept of *timing abstraction* is present, formally the notion of *time itself* is often not. Also, in pipelined and superscalar processors: are specification state components distributed in time in an implementation [17]; or (our position) are they *functions* of implementation state components from the *same* time? Consider a three-stage pipeline in which instructions are fetched three cycles ahead of execution⁴. Is the the program counter *pc* in the specification the value from the implementation three [specification] cycles earlier, or the current value less three⁵? This always enables us to separate timing and data abstraction maps. Recent work addresses larger and more complex examples: the VAMP project, [1], the ARM6 verification project at Cambridge[5, 6], Hunt’s Group Austin,Texas[15], and the UV group at Utah [13].

2 Clocks and Basic Models of Computers

A clock T is an algebra $T = (\mathbf{N} \mid 0, t + 1)$ denoting intervals of time called *clock cycles*. Time is defined in terms of events and not vice versa: typically, clock cycles mark the beginning/end of ‘interesting’ events, and need not be equal in length. In addition, clock cycles identify *intervals* of time, and not *points* in time.

Systems are modelled by *iterated maps* $F : T \times A \rightarrow A$, where A is a state set. St is implemented in terms of *next-state* and *initialization* functions:

$$\begin{aligned} F(0, a) &= \text{init}(a), \\ F(t + 1, a) &= \text{next}(F(t, a)). \end{aligned}$$

State set A is a Cartesian product of simpler state components. In general, we expect machine algebra operations to at most [simultaneous] primitive recursive functions. Hence F is generally a simultaneous primitive recursive function.

2.1 Initialization Functions in Iterated Map Models

The rôle of initialization functions is *not* to describe the initial behaviour of a system. Rather it is to eliminate unwanted *starting states in traces*. The choice of initialization function will vary according to circumstances.

²At least with reference to the state-of-the-art in processor verification at the time.

³And also work on verifying processor *fragments*.

⁴Neglecting complications to do with branching and so on.

⁵More likely, some multiple of three.

However, our usually-preferred initialization function $init : A \rightarrow A$ leaves initial state $a \in A$ unchanged *provided* a is already consistent with correct future state traces of F . Such initialization functions are an important part of the verification process (Sections 2.1 and 3), and, together with duration functions, are analogous to the *pipeline invariants* of [3] and others. Constructing suitable initialization functions can be difficult: [4, 9] describe a systematic technique that can be used whenever the contents of a pipeline are uniquely determined at time s by its contents at time s . This is not always the case - consider an example with two integer execution units, in which the unit chosen for a particular instructions is determined by which has the shortest queue: the contents of the queues may be a function of instructions that have already left the pipeline.

3 Correctness Models for Non-Pipelined, Pipelined and Superscalar Processors

Correctness models relate iterated maps $F : T \times A \rightarrow A$ and $G : S \times B \rightarrow B$. The majority of effort in the model described here is devoted to time: the relationship between state sets A and B is a typically a straightforward projection $\pi : B \rightarrow A$. However, timing abstraction is complex.

A *timing abstraction map* $\lambda : S \rightarrow T$ is a surjective (all specification times occur in the implementation) and monotonic (time does not go backwards) map. Timing abstraction maps are typically parameterized by the implementation state, and since microprocessors are deterministic, λ is uniquely determined by the initial implementation state.

$$\lambda : B \rightarrow [S \rightarrow T]$$

Each timing abstraction map λ possesses a corresponding *immersions* $\bar{\lambda} : T \rightarrow S$:

$$\bar{\lambda}(t) = \text{least } s \mid \lambda(s) \geq t$$

and the *start* operator $start : [S \rightarrow T] \rightarrow [S \rightarrow S]$

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

Microprocessors can be modelled at different levels of abstraction. Of concern here are the lowest level accessible by a programmer: termed the *programmer's model* PM in this paper, and the most abstract implementation level: termed the *abstract circuit model* AC . Clock cycles in PM models correspond with machine instructions: in AC models they are [some multiple of] system clock cycles. A non-pipelined microprocessor implementation G of AC is correct with respect to a specification F of PM if and only if the state of G under data abstraction map π is identical to the state of F for all times $s \in S$ corresponding with the start/end of cycles of T . That is, for all $s = start(\lambda)(s)$:

$$F(\bar{\lambda}(b)(s), \pi(b)) = \pi(G(s, b)). \tag{1}$$

In a pipelined processor, instruction execution overlaps, and *during instruction execution* we cannot uniquely relate cycles of S with cycles of T . However, instructions *terminate* at unique times: If instruction i terminates at time s_i , then no other instruction will terminate at time s_i . Provided timing abstraction map λ relates s to the time $t_i \in T$ corresponding with the end of instruction i and the start of $i+1$, the correctness statement in equation (1) still applies [9].

Superscalar processors attempt to execute multiple [pipelined] instructions in parallel, and instructions are allowed to terminate simultaneously, or out of program order. It is not possible to uniquely associate cycles of S corresponding with the start/end of instructions with cycles of T . The approach taken is based on

the following: in the event that instructions i and $i + 1$ terminate simultaneously or out of order, it is not meaningful to ask ‘is G correct with respect to F after i has terminated but before $i + 1$ ’ because there is no such time.

A new *retirement clock* R marks the completion of one or more instructions, with timing abstraction maps $\lambda_1 : T \rightarrow R$ and $\lambda_2 : S \rightarrow R$. timing abstraction map λ_1 captures the relationship between the sequential ‘one-at-a-time’ execution model of the programmer and the actual order of instruction completion; λ_2 marks instruction completion times with respect to the system clock. The non-surjective *adjunct timing abstraction map* $\rho : S \rightarrow T$ constructed by

$$\rho(s) = \bar{\lambda}_1 \lambda_2(s)$$

relates system clock times and the completion of machine instructions.

The correctness statement in equation (1) still applies if timing abstraction map λ is replaced with adjunct timing abstraction map ρ [4].

4 One-Step Theorems

Given the presence of an explicit clock, the obvious correctness proof for the models in section 2.1 is induction over clock S . However, induction is not necessary if the following two conditions are met.

- Iterated map $G : S \times B \rightarrow B$ is *time-consistent*. That is:

$$G(s, b) = \text{init}_G(G(s, b))$$

for all $s = \text{start}(\lambda)(s)$ and where init_G is the initialization function for iterated map G .

- timing abstraction map λ is *uniform*. That is for all $t \in T$

$$\bar{\lambda}(b)(t + 1) - \bar{\lambda}(b)(t) = \text{dur}(b),$$

where $\text{dur} : B \rightarrow \mathbf{N}^+$ is a *duration function* (see below).

The conditions above establish the independence of AC model G from the numerical value of $s \in S$. As well as being necessary conditions for the application of the one-step theorems, time-consistency and uniformity are characteristics of real hardware, so models that did not possess them would be unconvincing.

To establish that timing abstraction map λ is uniform, it is sufficient to define its immersion in terms of a duration function $\text{dur} : B \rightarrow \mathbf{N}^+$:

$$\begin{aligned} \bar{\lambda}(b)(0) &= 0, \\ \bar{\lambda}(b)(t + 1) &= \text{imm}(b)(t) + \text{dur}(G(\bar{\lambda}(b)(t), b)). \end{aligned}$$

Because a typical implementation G is extremely complex, defining dur independently of G is usually prohibitively difficult. The usual definition is *non-constructive* of the form:

$$\text{dur}(b) = \text{least } s \mid \text{end}(G(s, b)),$$

where $\text{end} : B \rightarrow \mathbf{B}$ is some function that identifies when one (or more) instructions have completed. Because G forms part of the definition of dur and λ , in this case the correctness model makes no statement about how long each instruction will take to execute.

Initially, it seems that establishing time-consistency requires induction. However, the first one-step theorem addresses this. Given iterated map $G : S \times B \rightarrow B$, and timing abstraction map $\lambda : S \rightarrow [S \rightarrow T]$ then to establish

$$G(s, b) = \text{init}_G(G(s, b))$$

for all $s = \text{start}(\lambda)(s)$, it is sufficient to show that:

$$\begin{aligned} G(0, b) &= \text{init}_G(G(0, b)), \text{ and} \\ G(\bar{\lambda}(b)(1), b) &= \text{init}(G(\bar{\lambda}(b)(1), b)). \end{aligned}$$

The proof [4] is omitted.

The second one-step theorem can be used to establish correctness. Given time-consistent iterated maps $F : S \times A \rightarrow A$ and $G : S \times B \rightarrow B$, and uniform timing abstraction map $\lambda : S \rightarrow [S \rightarrow T]$ then to establish

$$F(\bar{\lambda}(b)(s), \pi(b)) = \pi(G(s, b))$$

for all $s = \text{start}(\lambda)(s)$, it is sufficient to show that:

$$\begin{aligned} F(0, \pi(b)) &= \pi(G(0, b)), \text{ and} \\ F(1, \pi(b)) &= \pi(G(\bar{\lambda}(b)(1), b)). \end{aligned}$$

The proof [4] is omitted.

Discussion of the superscalar case where F and G are related by an adjunct timing abstraction map ρ is omitted, other than to say that the one step theorems still hold [4, 9].

5 VTM Model Definition

In this section, the existing microprocessor model is extended to accommodate SMT/CMT. From the perspective of an operating system kernel programmer, an SMT/CMT processor appears as multiple *PM*-level processors in which some state is *shared*. These processors will be implemented, collectively, by a single *AC*-level model. The term *Virtual Thread Model (VTM)* is used to distinguish these processors from the conventional *PM* level.

The temporal relationship with *other VTM* processors is exposed via the shared state. This relationship is defined by state information that is *not* present in the *VTM* state, but is in the *implementation (AC)* state. For example, one implementation may choose to prioritize one thread at the expense of others as a function of state elements not visible at the *VTM* level, while another implementation *of the same VTM level model* may not. Consequently, *VTM* models must be defined over a *PM* state set extended by at least part of the corresponding *AC* state. In this paper the complete *AC* state is used. However, there is a case for introducing a new, intermediate, level of abstraction [10].

Consider a Simultaneous Multi-Threaded/Multi-Core processor that is able to execute n threads - that is, it appears to be n (virtual) processors. Each virtual processor F_{VTM}^i , $i \in \{1, \dots, n\}$, will operate over its own clock T_i ; the state of F_{VTM}^i will be composed of some parts that are *local* to F_{VTM}^i and some parts that are *shared* with $F_{\text{VTM}}^1, \dots, F_{\text{VTM}}^{i-1}, F_{\text{VTM}}^{i+1}, \dots, F_{\text{VTM}}^n$. We assume, without loss of generality, that the private state elements $\text{priv} \in \Sigma_{\text{VTM}}^{\text{priv}}$ precede the shared state elements $\text{share} \in \Sigma_{\text{VTM}}^{\text{share}}$ in the state vector:

$$\Sigma_{\text{VTM}} = \Sigma_{\text{VTM}}^{\text{priv}} \times \Sigma_{\text{VTM}}^{\text{share}}$$

The state trace of each *VTM* processor will be a function of its own local and shared state, and the shared state of all other *VTM* processors. There is only one shared state in the *AC* level implementation. However

each individual *VTM* model has its own copy of the shared state, from its own perspective: a conceit it is convenient to maintain. Consequently it is necessary to merge the shared states of each *VTM* level processor.

Each *VTM* processor operates with its own clock: to correctly merge shared states from different *VTM* processors, we must match states at the appropriate times. Times on different *VTM* processors are related using the [adjunct] timing abstraction maps and corresponding immersions between *VTM* and *AC* level clocks: $\rho_i(b)\bar{\rho}_j(b)(t_j)$ is the time on clock T_i corresponding to time $t_j \in T_j$.

Given clocks $T_i, i \in \{1, \dots, n\}$; F_{ITM} state set Σ_{AC} , F_{VTM} state set Σ_{VTM} ; private state projection functions $\pi_{\text{priv}}^i : \Sigma_{\text{VTM}} \rightarrow \Sigma_{\text{VTM}}^{\text{priv}}$ for $i \in \{1, \dots, n\}$; merge operators $\tau_i : (\Sigma_{\text{VTM}})^n \rightarrow \Sigma_{\text{VTM}}^{\text{share}}$; data abstraction functions $\psi^i : \Sigma_{\text{AC}} \rightarrow \Sigma_{\text{VTM}}$ for $i \in \{1, \dots, n\}$ and next-state and initialization functions $next : \Sigma_{\text{VTM}} \rightarrow \Sigma_{\text{VTM}}$ and $init : \Sigma_{\text{VTM}} \rightarrow \Sigma_{\text{VTM}}$, an individual F_{VTM} level processor F_{VTM}^i is modeled as follows.

$$\begin{aligned}
& F_{\text{VTM}}^i : \Sigma_{\text{AC}} \rightarrow [T_i \times \Sigma_{\text{VTM}} \rightarrow \Sigma_{\text{VTM}}] \\
& F_{\text{VTM}}^i(\sigma_{\text{AC}})(0, \overrightarrow{\sigma_{\text{VTM}}}) = init(\overrightarrow{\sigma_{\text{VTM}}}) \\
& F_{\text{VTM}}^i(\sigma_{\text{AC}})(t + 1, \overrightarrow{\sigma_{\text{VTM}}}) = \tag{2} \\
& \quad next[\pi_{\text{priv}}^i(F_{\text{VTM}}^i(\sigma_{\text{AC}})(t, \overrightarrow{\sigma_{\text{VTM}}})], \\
& \quad \tau_i(F_{\text{VTM}}^i(\sigma_{\text{AC}})(t, \overrightarrow{\sigma_{\text{VTM}}}), \\
& \quad F_{\text{VTM}}^1(\sigma_{\text{AC}})(\rho_i(\sigma_{\text{AC}})\bar{\rho}_1(\sigma_{\text{AC}})(t), \psi_1(\sigma_{\text{AC}})), \\
& \quad \vdots \\
& \quad F_{\text{VTM}}^{i-1}(\sigma_{\text{AC}})(\rho_i(\sigma_{\text{AC}})\bar{\rho}_{i-1}(\sigma_{\text{AC}})(t), \psi_{i-1}(\sigma_{\text{AC}})), \\
& \quad F_{\text{VTM}}^{i+1}(\sigma_{\text{AC}})(\rho_i(\sigma_{\text{AC}})\bar{\rho}_{i+1}(\sigma_{\text{AC}})(t), \psi_{i+1}(\sigma_{\text{AC}})), \\
& \quad \vdots \\
& \quad F_{\text{VTM}}^n(\sigma_{\text{AC}})(\rho_i(\sigma_{\text{AC}})\bar{\rho}_n(\sigma_{\text{AC}})(t), \psi_n(\sigma_{\text{AC}}))].
\end{aligned}$$

F_{VTM}^i is defined over both the *PM*-level state set Σ_{VTM} and the *AC*-level state set Σ_{AC} . Since we would normally expect that $\Sigma_{\text{VTM}} \subset \Sigma_{\text{AC}}$, this may seem unnecessary: however, expressing the definition of F_{VTM}^i in this form is helpful in formally establishing the one-step theorems still hold [11].

Note that as well as containing the *AC*-level state Σ_{AC} , our *VTM*-level definition contains the state dependent adjunct timing abstraction maps ρ_i . Recall that it is usual to define ρ_i in terms of some *AC*-level implementation (Section 3). Consequently, the *AC* implementation is deeply embedded in the definition of a *VTM* level model.

A significant issue in the *VTM* level model is that next-state and initialization functions from a [probably existing] *PM* level model. Since the definitions of $next$ and $init$ are by far the most complex part of model definition, it is important to be able to reuse them.

The definition above is the most general case: in some circumstances it can be simplified, depending on the requirements of the merge operators τ_i that unify the various shared state components of the n *VTM* processors. The definition of τ will depend on the precise nature of the shared state $\Sigma_{\text{VTM}}^{\text{share}}$; and the behaviour of the processor implementation - for example, if two *VTM* processors attempt to update the same state unit simultaneously. Commonly, the shared state will consist of the processor's main memory. In some circumstances, the definitions of τ_i will not be functions of the private state; and in others all the merge operators τ_i are identical: see [10, 11].

5.1 Correctness of the VTM Model

What does it mean for a *VTM* level model to be correctly implemented by an *AC* level model? Note that because there are n *VTM* level processors corresponding to each *AC* level processor, there are n separate correctness statements.

Intermediate Thread Model map $F_{\text{ITM}} : S \times \Sigma_{\text{AC}} \rightarrow \Sigma_{\text{AC}}$ is said to be a *correct implementation* of Virtual Thread Model maps $F_{\text{VTM}}^i : \Sigma_{\text{AC}} \rightarrow [T_i \times \Sigma_{\text{VTM}} \rightarrow \Sigma_{\text{VTM}}]$, $i \in \{1, \dots, n\}$ for some initializations if, given state-dependent adjunct retimings $\rho_i \in \text{Ret}(\Sigma_{\text{AC}}, S, T_i)$ and surjective data abstraction maps $\psi_i : \Sigma_{\text{AC}} \rightarrow \Sigma_{\text{VTM}}$, $i \in \{1, \dots, n\}$ then, for each clock T_i , $\forall s = \text{start}(\rho_i(\sigma_{\text{AC}}))(s)$ and $\text{state} \in \Sigma_{\text{AC}}$, then

$$\begin{aligned} \psi_i(F_{\text{ITM}}(s, \sigma_{\text{AC}})) = \\ F_{\text{VTM}}^i(\sigma_{\text{AC}})(\rho_i(\sigma_{\text{AC}})(s), \psi_i(\sigma_{\text{AC}})). \end{aligned}$$

In considering the application of the one-step theorems to the VTM model, recall that the *AC*-level processor $G : S \times B \rightarrow B$ is essentially identical to any other *AC* processor model: that is, it represents the implementation of a more abstract specification and hence the existing one-step theorems show how to establish that G is time-consistent. Clearly it is possible to establish the uniformity of timing abstraction map ρ_i by construction in terms of a duration function (section 2.1). However, time-consistency of the collection of *VTM* level processors F_i , $i \in \{1 \dots n\}$ must be established. The corresponding theorem and proof are omitted here but can be found in [11, 10].

6 AC-P²: A Dual-Core Pipelined Example.

SPM is a simple, five-instruction RISC architecture, with pipelined and superscalar implementations AC-P and AC-S [4, 8]. In this section, the model described above is illustrated by a dual-core version of AC-P, called AC-P² which simply duplicates the instruction pipeline of AC-P (that is, all state components except for data and program memory).

6.1 The SPM Architecture

The SPM architecture consists of a program memory pm , data memory dm , register set r and program counter pc , parameterized by w , m and r which are respectively the wordsize, memory address size and register index.

$$\Sigma_{\text{SPM}} = \text{Mem}^2 \times \text{MAR} \times \text{Reg},$$

where $\text{MAR} = W_m$, $\text{Word} = W_w$, $\text{RI} = W_r$, $\text{Reg} = [\text{RI} \rightarrow \text{Word}]$, $\text{Mem} = [\text{MAR} \rightarrow \text{Word}]$, and W_i is the set of words of length i . The five instructions are:

- **add ra rb rc** - $rc = ra + rb$; $pc = pc + 1$;
- **branch addr** - *if* $r0 == 0$ *then* $pc = pc + \text{addr}$ *else* $pc = pc + 1$;
- **load ra addr** - $r[\text{ra}] = \text{dm}[\text{addr}]$; $pc = pc + 1$;
- **store ra addr** - $\text{dm}[\text{addr}] = r[\text{ra}]$; $pc = pc + 1$;
- **set ra val** - $r[\text{ra}] = \text{val}$; $pc = pc + 1$.

We do not explicitly specify the various instruction formats other than to state that there are fields for *opcode*, *register indices* r_a , r_b and r_c , and *addresses/immediate values*. The address/immediate value fields are identical, and they may overlap the r_b and r_c fields since they are not needed simultaneously.

SPM is formally defined as follows.

$$\begin{aligned} SPM &: T \times \Sigma_{\text{SPM}} \rightarrow \Sigma_{\text{SPM}}, \\ SPM(0, pm, dm, pc, r) &= (pm, dm, pc, r), \\ SPM(t + 1, pm, dm, pc, r) &= \\ &\quad spm(SPM(t, pm, dm, pc, r)), \end{aligned}$$

where $spm : \Sigma_{\text{SPM}} \rightarrow \Sigma_{\text{SPM}}$ is defined by

$$spm(pm, dm, pc, r) = \left\{ \begin{array}{ll} (pm, dm, pc + 1, & \text{if } opr = \text{add}; \\ r[r_a(ins) + r_b(ins)/r_c(ins)], & \\ (pm, dm, pc + addr(ins), r), & \text{if } opr = \text{branch} \\ & \text{and } r[0] = 0; \\ (pm, dm, pc + 1, r), & \text{if } opr = \text{branch} \\ & \text{and } r[0] \neq 0; \\ (pm, dm, pc + 1, & \text{if } opr = \text{load}; \\ r[dm[addr(opr)]/r_a(ins)], & \\ (pm, dm[r_a(opr)]/addr(ins), & \text{if } opr = \text{store}; \\ pc + 1, r), & \\ (pm, dm, pc + 1, & \text{if } opr = \text{set}, \\ r[pad(val(opr))/r_a(opr)], & \end{array} \right.$$

where $ins = pm[pc]$, $opc = op(ins)$, op , r_a, r_b, r_c and $addr = val$ are the obvious projections, and $pad : MAR \rightarrow Word$ pads with leading zeros.

6.2 AC-P² Implementation Overview

The AC-P pipeline on which AC-P² is based has a four stage pipeline (fetch, decode, execute, commit), and seven component parts, corresponding to the pipeline stages (except commit) with the addition of instruction memory, data memory, registers and program counter. AC-P² duplicates all components except instruction and data memory: see fig 1. Note that the definition of AC-P² here is incomplete. The full definition is to be found in the appendix. Resource conflict issues (e.g. multiple simultaneous memory/register accesses) are neglected in AC-P² (as they are in the non-CMT version AC-P [4]). However, AC-P² does correctly manage read-after-write (RAW) hazards and procedural dependencies. The pipeline is managed by a small counter ctr and a state element $unit$ that determines the action/destination of instruction results - one of the possible values of which is *wait*, preventing instruction results being committed. The pipeline can be put into a *flushed* state by setting $unit = wait$ and $ctr = 2$. The value of ctr is decremented each cycle and results are only committed when $ctr = 0$ and $unit \neq wait$. Immediately after a flush state the pipeline will move into an *after branch* state (so-called because it will also enter this state after a successful conditional branch) with $ctr = 1$ and $unit = wait$; then into an *after conflict* state (it can also enter this state after a RAW hazard) with $ctr = 0$ and $unit = wait$; then into a *pipe full* state with $ctr = 0$ and $unit \neq wait$ (that is, permitting an instruction result to commit).

The behaviour of AC-P² is defined in terms of iterated maps for each of the fetch, decode, execute, register,

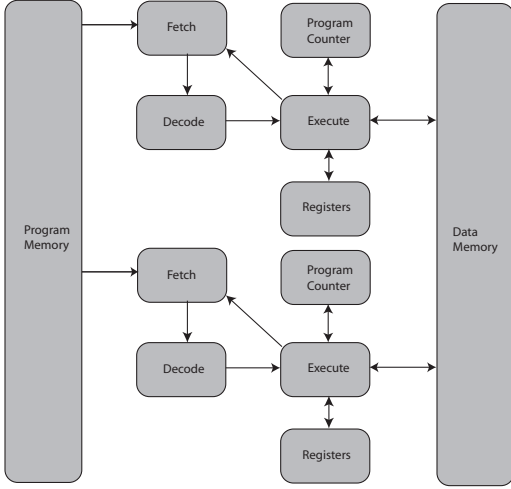


Figure 1: The Structure of AC-P².

program counter and data memory units (instruction memory never changes): in the final definition, all iterated maps except data memory are duplicated.

6.3 AC-P² State

The state of the fetch, decode and execute units is as follows (the register, program counter and data memory units each contain only the registers, program counter and data memory respectively).

- **Fetch unit** - $Ftch = Word \times MAR$. The fetch unit contains an instruction register and a fetch program counter.
- **Decode unit** - $Dec = Op \times RI^3 \times MAR$, where $Op = W_3$. The decode unit contains the component parts of the instruction.
- **Execute unit** - $Ex = Word \times MAR \times Unit \times Ctr$, where $Unit = \{reg, pc, incpc, dmem, wait\}$ and $Ctr = W_2$. The execute unit contains a result word and an address word, specifying a result's final destination. Note the (plausible) assumption that RI is no larger than MAR ($r \leq m$) and hence a register index will fit in a memory address register. It is also assumed that $MAR \leq Word$. $Unit$ represents the action/destination of an instruction, and Ctr manages pipeline flushing.

The state-set of AC-P² is hence:

$$\Sigma_{ac-p^2} = (Ftch \times Dec \times Ex \times Reg \times MAR)^2 \times Mem^2$$

where $\Sigma_{ac-p^2}^{share} = Mem^2$ represents the shared state and $\Sigma_{ac-p^2}^{priv} = Ftch \times Dec \times Ex \times Reg \times MAR$ the private state of each pipeline. Note that $\Sigma_{ac-p^2} = \Sigma_{ac-p^2}^{priv}{}^2 \times \Sigma_{ac-p^2}^{share}$.

6.4 Definition of AC-P²

We define AC-P² as follows.

$$\begin{aligned} ACP^2 &: S \times \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}, \\ ACP^2(0, \sigma) &= \mathit{init}(\sigma), \\ ACP^2(s+1, \sigma) &= \mathit{next}(ACP^2(s, \sigma)), \end{aligned}$$

where $\mathit{next} : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}$ is defined by

$$\begin{aligned} \mathit{next}(p1, p2, pc2, dm, pm) &= \mathit{fetch}(p1, dm, pm), \\ &\mathit{decode}(p1, dm, pm), \mathit{execute}(p1, dm, pm), \\ &\mathit{registers}(p1, dm, pm), \mathit{progcount}(p1, pm, dm), \\ &\mathit{fetch}(p2, dm, pm), \mathit{decode}(p2, dm, pm), \\ &\mathit{execute}(p2, dm, pm), \\ &\mathit{registers}(p2, dm, pm), \mathit{progcount}(p2, pm, dm), \\ &\mathit{datamem}(p1, p2, dm, pm), pm), \end{aligned}$$

and $p1, p2 \in \Sigma_{ac-p^2}^{\text{priv}}$. The initialization function init need only put AC-P² into some legal state to be minimally correct. However, to make use of the one-step theorems, init should only modify the state of AC-P² if it not legal. The full definition is omitted (but see the Appendix), but briefly init performs a reset (flush) and then progressively refills the pipeline. If the pipeline state is correct, then at some point the (refilled) pipeline state should match the original. For example, here is the definition of $\mathit{pipefull}_1 : \Sigma_{ac-p^2} \rightarrow \mathbf{B}$ which determines if pipeline 1 is in a full state:

$$\mathit{pipefull}_1(\sigma) = \begin{cases} \mathit{tt}, & \text{if } \pi_1(\sigma) = \pi_1(\mathit{next}^3(\mathit{flush}(\sigma))); \\ \mathit{ff}, & \text{otherwise,} \end{cases}$$

where $\pi_1 : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}^{\text{priv}} \times \Sigma_{ac-p^2}^{\text{share}}$ projects the private state of pipeline 1 and the shared state.

where flush flushes a pipeline by simply setting $\mathit{unit} = \mathit{wait}$ and $\mathit{ctr} = 2$. There are similar functions for the *after branch* and *after conflict* states.

It remains to define the next-state functions fetch , decode , $\mathit{execute}$, $\mathit{registers}$, $\mathit{progcount}$ and $\mathit{datamem}$, as well as $\mathit{conflict} : \Sigma_{ac-p^2} \rightarrow \mathbf{B}$ which establishes if the pipeline must stall. For the definitions of these functions, see the Appendix.

6.5 VTM-Level Model

In order to complete the VTM-level definition (and neglecting trivial projection operators), it remains to define the timing abstraction maps $\lambda_{1,2}$, the *merge* operators $\tau_{1,2} : \Sigma_{\text{SPM}}^2 \rightarrow \Sigma_{\text{SPM}}^{\text{share}}$, where $\Sigma_{\text{SPM}}^{\text{share}} = \mathit{Mem}$, and the data abstraction maps $\psi_{1,2} : \Sigma_{ac-p^2} \rightarrow \Sigma_{\text{SPM}}$.

Note that AC-P² is pipelined but not superscalar, so the timing abstraction maps are not adjunct. The timing abstraction functions $\lambda_{1,2}$ are defined in terms of their duration functions, since $\lambda_{1,2}$ are trivially constructed from them (section 3). Duration function $\mathit{dur}_1 : \Sigma_{ac-p^2} \rightarrow \mathbf{N}^+$ for pipeline 1 is defined by

$$\mathit{dur}_1(\sigma) = \begin{cases} 1, & \text{if } \mathit{pipefull}_1(\sigma); \\ 2, & \text{if } \mathit{afterconflict}_1(\sigma); \\ 3, & \text{if } \mathit{afterbranch}_1(\sigma); \\ 4, & \text{otherwise.} \end{cases}$$

The definitions of the merge operator and data abstraction map for pipeline 1 are as follows.

$$\begin{aligned} \tau_1((m1, pc1, r1), (m2, pc2, r2))[i] = & \\ \begin{cases} m1[i], & \text{if } m1[i] = m2[i]; \\ m2[i], & \text{if } m1[i] \neq m2[i]. \end{cases} & \\ \psi_1(f1, d1, e1, pc1, r1, f2, d2, e2, pc2, r2, dm, pm) = & \\ (pm, dm, pc1, r1). & \end{aligned}$$

Notice that $\tau_{1,2}$ are dependent on only the shared state in this case. A more restricted form of the VTM definition in Section 5 is possible [10] but is omitted here.

The definition of the VTM model for pipeline 1 is as follows.

$$\begin{aligned} VTM : \Sigma_{ac-p^2} &\rightarrow [T_1 \times \Sigma_{SPM}] \rightarrow \Sigma_{SPM}, \\ VTM(\sigma_{acp})(0, \sigma_{spm}) &= \sigma_{spm}, \\ VTM(\sigma_{acp})(t+1, \sigma_{spm}) &= \\ &spm[\pi_{priv}(VTM_1(\sigma_{acp})(t, \sigma_{spm})), \\ &\tau_1(VTM_1(\sigma_{acp})(t, \sigma_{spm})), \\ &VTM_2(\sigma_{acp})(\lambda_2(\sigma_{acp})\bar{\lambda}_1(\sigma_{acp}), \psi_2(\sigma_{acp}))]. \end{aligned}$$

7 Concluding Remarks

And existing model of microprocessors and their correctness has been extended to superscalar SMT and CMT processor implementations, which represent the state-of-the-art in current commercial implementation. However, although it is possible to successfully model such processors, and define what it means for them to be correct, practical verification of realistic examples would be a formidable undertaking at the present time. Nonetheless, there is a case for developing models of processors and their correctness that run ahead of useful application: the modeling approaches to pipelined processors that were ultimately used to verify ARM6 [5, 6] were developed some years in advance of their practical use.

A point worthy of note is the presence of the definition of the *AC* level implementation in the definition of the *VTM* level model. This should not be surprising: in practice, the implementation of an SMT or multi-core processor *does* impact the behaviour seen by programmers; and the timing behaviour of all processors is a function of their implementation. This last fact is generally acknowledged in our model by the definition of timing abstraction maps in terms of the *AC*-level model. There has been a general weakening of the long-established separation of processor architecture and implementation.

References

- [1] S Beyer, C Jacobi, D Kröning, D Leinenbach, and W Paul. Instantiating uninterpreted functional unit and memory system: Functional verification of VAMP. In D Geist and T Enrico, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2003.
- [2] J Burch and D Dill. Automatic verification of pipelined microprocessor control. In D Dill, editor, *Proceedings of the 6th International Conference, CAV'94: Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994.

- [3] D Cyrluk, J Rushby, and M Srivas. Systematic formal verification of interpreters. In *IEE international conference on formal engineering methods ICFEM'97*, pages 140–149, 1997.
- [4] A C J Fox. *Algebraic Representation of Advanced Microprocessors*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1998.
- [5] A C J Fox. Formal specification and verification of ARM6. In D Basin and B Wolff, editors, *TPHOLs '03*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer-Verlag, 2003.
- [6] A C J Fox. An algebraic framework for verifying the correctness of hardware with input and output: a formalization in HOL. In J L Fiadeiro, N A Harman, M Roggenbach, and J Rutten, editors, *CALCO 2005*, volume 3629 of *Lecture Notes in Computer Science*, pages 157–174. Springer-Verlag, 2005.
- [7] 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*, volume 1546 of *Lecture Notes in Computer Science*, pages 138–183. Springer-Verlag, 1998.
- [8] A C J Fox and N A Harman. Algebraic models of correctness for microprocessors. *Formal Aspects of Computing*, 12(4):298–312, 2000.
- [9] A C J Fox and N A Harman. Algebraic models of correctness for abstract pipelines. *The Journal of Algebraic and Logic Programming*, 57:71–107, 2003.
- [10] N A Harman. Algebraic models of simultaneous multi-threaded and chip-level multi-threaded microprocessors. Submitted to the Journal of Algebraic and Logic Programming, 2007.
- [11] N A Harman. Algebraic models of simultaneous multithreaded and multi-core processors. Proceedings of CALCO 07, Springer-Verlag Lecture Notes in Computer Science, to appear CSR6-2007, University of Wales Swansea, Computer Science Department, <http://cs.swan.ac.uk/reports/yr2007/CSR6-2007.pdf> or <http://www-compsci.swan.ac.uk/~csneal/Reports/CSR6-2007.pdf>, 2007.
- [12] N A Harman. Modelling SMT and CMT processors: A simple case study. Technical Report CSR7-2007 (Submitted to FMCAD 07), University of Wales Swansea, Computer Science Department, <http://cs.swan.ac.uk/reports/yr2007/CSR7-2007.pdf> or <http://www-compsci.swan.ac.uk/~csneal/Reports/CSR7-2007.pdf>, 2007.
- [13] R Hosabettu, G Gopalakrishnan, and M Srivas. Formal verification of a complex pipelined processor. *Formal Methods in System Design*, 23(2):171–213, 2003.
- [14] 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.
- [15] S Ray and W A Hunt. Deductive verification of pipelined machines using first-order quantification. In *Computer-Aided Verification CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 31–43. Springer-Verlag, 2004.
- [16] K Stephenson. Algebraic specification of the Java virtual machine. In B Möller and J V Tucker, editors, *Prospects for Hardware Foundations Foundations*, volume 1546 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
- [17] P Windley and J Burch. Mechanically checking a lemma used in an automatic verification tool. In A Camilleri and M Srivas, editors, *Formal methods in Computer-Aided Design computer-aided design*, volume 1166 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1996.
- [18] 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: Omitted AC-P² Definitions.

This appendix contains a more complete definition of the AC-P² implementation than that the main text. Definitions that are trivial, or can be trivially constructed from those below, are not included.

7.1 State Set

- **Fetch unit** - $Ftch = Word \times MAR$. The fetch unit contains an instruction register and a fetch program counter.
- **Decode unit** - $Dec = Op \times RI^3 \times MAR$, where $Op = W_3$. The decode unit contains the component parts of the instruction.
- **Execute unit** - $Ex = Word \times MAR \times Unit \times Ctr$, where $Unit = \{reg, pc, incpc, dmem, wait\}$ and $Ctr = W_2$. The execute unit contains a result word and an address word, specifying a result's final destination. Note the (plausible) assumption that RI is no larger than MAR ($r \leq m$) and hence a register index will fit in a memory address register. It is also assumed that $MAR \leq Word$. $Unit$ represents the action/destination of an instruction, and Ctr manages pipeline flushing.
- **Registers** - the register unit contains the single state element $[RI \rightarrow Word]$.
- **Program Counter** - the program counter unit contains the single state element MAR .
- **Data Memory and Program Memory** - the data and program memory units each contain the single state elements $[MAR \rightarrow Word]$.

The state-set of AC-P² is hence

$$\Sigma_{ac-p^2} = (Ftch \times Dec \times Ex \times Reg \times MAR)^2 \times Mem^2$$

where

$$\begin{aligned} Ftch &= Word \times MAR, \\ Dec &= Op \times RI^3 \times MAR, \\ Ex &= Word \times MAR \times Unit \times Ctr, \end{aligned}$$

and $\Sigma_{ac-p^2}^{share} = Mem^2$ represents the shared state and $\Sigma_{ac-p^2}^{priv} = Ftch \times Dec \times Ex \times Reg \times MAR$ the private state of each pipeline. Note that $\Sigma_{ac-p^2} = \Sigma_{ac-p^2}^{priv}{}^2 \times \Sigma_{ac-p^2}^{share}$.

7.2 Definition of AC-P²

We define AC-P² as follows.

$$\begin{aligned} ACP^2 &: S \times \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}, \\ ACP^2(0, \sigma) &= init(\sigma), \\ ACP^2(s+1, \sigma) &= next(ACP^2(s, \sigma)), \end{aligned}$$

where $next : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}$ is defined by

$$\begin{aligned} next(p1, p2, pc2, dm, pm) = & fetch(p1, dm, pm), decode(p1, dm, pm), execute(p1, dm, pm), \\ & registers(p1, dm, pm), progcount(p1, pm, dm), \\ & fetch(p2, dm, pm), decode(p2, dm, pm), execute(p2, dm, pm), \\ & registers(p2, dm, pm), progcount(p2, pm, dm), \\ & datamem(p1, p2, dm, pm), pm), \end{aligned}$$

and $p1, p2 \in \Sigma_{ac-p^2}^{priv}$.

The initialization function $init$ is defined in terms of Boolean functions that test if each of the AC-P² pipelines is in one of four legal states: *pipeline full*; *after conflict* (immediately after a RAW hazard, one cycle after a taken conditional branch, or two cycles after a complete pipeline flush); *after branch* (immediately after a taken conditional branch or one cycle after a pipeline flush); or *flushed*. There are eight functions in total: four for each pipeline. The functions for pipeline 1 are defined below.

$$pipefull_1(\sigma) = \begin{cases} tt, & \text{if } \pi_1(\sigma) = \pi_1(next^3(flush(\sigma))); \\ ff, & \text{otherwise,} \end{cases}$$

where $\pi_1 : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share}$ projects the private state of pipeline 1 and the shared state.

$$\begin{aligned} & afterconflict_1 : \Sigma_{ac-p^2} \rightarrow \mathbf{B}, \\ & afterconflict_1(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2)dm, pm) = \\ & \begin{cases} tt, & \text{if } (f1, d1) = \pi_1^{fd}(next^2(flush(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2), dm, pm))) \\ & \text{and } u1 = wait \text{ and } c1 = 1; \\ ff, & \text{otherwise,} \end{cases} \end{aligned}$$

where $\pi_1^{fd} : \Sigma_{ac-p^2} \rightarrow Ftch \times Dec$ projects the fetch and decode state elements of pipeline 1.

$$\begin{aligned} & afterbranch_1 : \Sigma_{ac-p^2} \rightarrow \mathbf{B}, \\ & afterbranch_1(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2)dm, pm) = \\ & \begin{cases} tt, & \text{if } f1 = \pi_f(next(flush(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2)dm, pm))) \\ & \text{and } u1 = wait \text{ and } c1 = 0;; \\ ff, & \text{otherwise,} \end{cases} \end{aligned}$$

where $\pi_1^f : \Sigma_{ac-p^2} \rightarrow Ftch$ projects the fetch and decode state elements of pipeline 1.

$$\begin{aligned} & flushstate_1 : \Sigma_{ac-p^2} \rightarrow \mathbf{B}, \\ & flushstate_1(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2)dm, pm) = \\ & = \begin{cases} tt, & \text{if } u1 = wait \text{ and } c1 = 2;; \\ ff, & \text{otherwise,} \end{cases} \end{aligned}$$

(Note that there is a bug, or bugs, in the original descriptions in [4] of *afterbranch* and *afterconflict* where the entire (flushed, and partially refilled) pipeline state is compared with the original.)

The function *flush* empties both pipelines.

$$\begin{aligned} & flush : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2}, \\ & flush(f1, d1, (w1, m1, u1, c1), r1, pc1, f2, d2, (w2, m2, u2, c2)dm, pm) = \\ & (f1, d1, (w1, m1, wait, 2), r1, pc1, f2, d2, (w2, m2, wait, 2)dm, pm). \end{aligned}$$

Note we need to be able to individually identify the states of each pipeline because (a) they may both be in different legal states and (b) we need to know the individual states when defining the duration functions used to construct the timing abstraction maps. However, we also need to flush *both* pipelines prior to determining the (pre-flushed) status of *either* of them. Failure to do so would result in one pipeline continuing to operate and possibly writing to the shared state $dm \in Mem$.

$$init : \Sigma_{ac-p^2} \rightarrow \Sigma_{ac-p^2},$$

$$init(\sigma) = \begin{cases} \sigma, & \text{if } (flushstate_1(\sigma) \text{ or } afterbranch_1(\sigma) \text{ or } afterconflict_1(\sigma) \text{ or } pipefull_1(\sigma)) \text{ and} \\ & (flushstate_2(\sigma) \text{ or } afterbranch_2(\sigma) \text{ or } afterconflict_2(\sigma) \text{ or } pipefull_2(\sigma)); \\ flush(\sigma), & \text{otherwise.} \end{cases}$$

It remains to define the next-state functions *fetch*, *decode*, *execute*, *registers*, *progcoun*t and *datamem*, as well as *conflict* : $\Sigma_{ac-p^2} \rightarrow \mathbf{B}$ which establishes if the pipeline must stall.

The *fetch* stage deals with three cases: instruction conflicts (pipeline stalls); fetching from branch destination; and fetching the next sequential instruction.

$$fetch : \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share} \rightarrow Ftch,$$

$$fetch((ir, fpc), d, (result, dest, unit, ctr), r, pc, dm, pm) = \begin{cases} ir, fpc, & \text{if } conflict(d, (result, dest, unit, ctr)); \\ pm[dest], dest + 1, & \text{if } unit = pc; \\ pm[fpc], fpc + 1, & \text{otherwise.} \end{cases}$$

The *decode* stage deals with two cases: pipeline stalls and decoding the instruction held in the *fetch* stage (by simply extracting the relevant fields).

$$decode : \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share} \rightarrow Dec,$$

$$decode((ir, fpc), (opr, ra, rb, rc, address), e, r, pc, dm, pm) = \begin{cases} (opr, ra, rb, rc, address), & \text{if } conflict((opr, ra, rb, rc, address), e); \\ op(ir), r_a(ir), r_b(ir), addr(ir), & \text{otherwise.} \end{cases}$$

The *execute* stage deals with four cases, where the first two result in instruction execution (handled by sub-function *exec*). The remainder concern the pipeline states that can occur subsequent to instruction conflicts, taken conditional branches, and pipeline flushes.

$$execute : \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share} \rightarrow Ex,$$

$$execute(f, (opr, ra, rb, rc, address), (result, dest, unit, ctr), r, pc, dm, pm) = \begin{cases} exec((opr, ra, rb, rc, address), reg, pc, dm), & \text{if not } conflict((opr, ra, rb, rc, address) \\ & (result, dest, unit, ctr)) \text{ and } ctr = 0 \text{ and } unit \neq wait; \\ exec((opr, ra, rb, rc, address), reg, pc + 1, dm), & \text{if not } conflict((opr, ra, rb, rc, address) \\ & (result, dest, unit, ctr)) \text{ and } ctr = 0 \text{ and } unit = wait; \\ (result, dest, wait, 0), & \text{if } conflict((opr, ra, rb, rc, address), \\ & (result, dest, unit, ctr)) \text{ and } ctr = 0; \\ (result, dest, wait, ctr - 1), & \text{otherwise.} \end{cases}$$

(Note there is a bug in the original description in [4] where the second case above is missing.)

The *exec* operation handles the creation of execution tuples when instruction execution proceeds. The components of the triple are: the result, its destination address/index, its destination unit (registers, data memory or program counter) or action (wait or increment program counter) and the value of the pipeline state counter (zero except in the event of a taken conditional branch).

$$\begin{aligned}
 &exec : Dec \times Reg \times MAR \times Mem \rightarrow Ex, \\
 &exec((opr, ra, rb, rc, address), r, pc, dm) = \begin{cases} ra + rb, pad(rc), reg, 0, & \text{if } opr = add; \\ result, pc + addr, pc, 2, & \text{if } opr = branch \text{ and } r[0] = 0; \\ result, dest, incpc, 0, & \text{if } opr = branch \text{ and } r[0] \neq 0; \\ dm[address], pad(ra), reg, 0, & \text{if } opr = load; \\ ra, address, dmem, 0, & \text{if } opr = load; \\ pad(address), pad(ra), reg, 0, & \text{if } opr = set; \end{cases}
 \end{aligned}$$

The (overloaded) operators *pad* pad words with leading zeros.

The *registers* unit writes a result to the appropriate register if required.

$$\begin{aligned}
 ®isters : \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share} \rightarrow [RI \rightarrow Word, \\
 ®isters(f, d, (result, dest, unit, ctr), r, pc, dm, pm) = \begin{cases} r[result/trim(dest)], & \text{if } unit = reg; \\ r, & \text{otherwise.} \end{cases}
 \end{aligned}$$

The operation *trim* : *MAR* \rightarrow *RI* truncates the contents of *dest* (by removing leading zeros previously added by *pad*).

The *progcount* unit sets the program counter to one of the next sequential value or the branch destination address, or leaves it unchanged depending on the operation/pipeline state.

$$\begin{aligned}
 &progcount : \Sigma_{ac-p^2}^{priv} \times \Sigma_{ac-p^2}^{share} \rightarrow MAR, \\
 &progcount(f, d, (result, dest, unit, ctr), r, pc, dm, pm) = \begin{cases} dest, & \text{if } unit = pc; \\ pc, & \text{if } unit = wait; \\ pc + 1, & \text{otherwise;} \end{cases}
 \end{aligned}$$

The *datamem* unit writes a result to the appropriate memory word if required.

$$\begin{aligned}
 &datamem : \Sigma_{ac-p^2} \rightarrow Mem, \\
 &datamem((f1, d1, (res1, dest1, unit1, ctr1), r1, pc1), (f2, d2, (res2, dest2, unit2, ctr2), r2, pc2), dm, pm) = \\
 &\begin{cases} dm[res1/dest1], & \text{if } unit1 = dmem \text{ and } unit2 \neq dmem; \\ dm[res2/dest2], & \text{if } unit2 = dmem \text{ and } unit1 \neq dmem; \\ dm, & \text{otherwise.} \end{cases}
 \end{aligned}$$

Note that *datamem* does not specify what happens in the event that *dest1* = *dest*.

The *conflict* operation identifies if the pipeline must stall because of some instruction conflict.

$$\begin{aligned}
 & \text{conflict} : Dec \times Ex \rightarrow \mathbf{B}, \\
 & \text{conflict}(opr, ra, rb, rc, address), (result, dest, unit, ctr) = \\
 & \begin{cases} tt, & \text{if } unit = reg \text{ and } (opr = branch \text{ and } dest = 0); \\
 tt, & \text{if } op = add \text{ and } (trim(dest) = ra \text{ or } trim(dest) = rb); \\
 tt, & \text{if } op = store \text{ and } trim(dest) = ra; \\
 tt, & \text{if } op = load \text{ and } unit = dmem \text{ and } dest = address; \\
 ff, & \text{otherwise.} \end{cases}
 \end{aligned}$$

Informally, the cases are as follows.

- The decoded operation is a branch and the operation being executed is writing to register zero.
- The decoded operation is an add and the operation being executed is writing to one of its arguments.
- The decoded operation is a store and the operation being executed is writing to its destination address.
- The decoded operation is a load and operation being executed is writing to the memory word to be loaded.