Putting it all together – Formal Verification of the VAMP

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Sven Beyer

sbeyer@cs.uni-sb.de

Saarbrücken, März 2005

Tag des Kolloquiums:	18.03.2005
Dekan:	Prof. Dr. Jörg Eschmeier
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Holger Hermanns
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Wolfram Büttner
3. Berichterstatter:	Prof. Bernd Finkbeiner, PhD
akademischer Mitarbeiter:	Dr. Christian Lindig

Goodbye to you, my trusted friend.
We've known each other since we're nine or ten.
Together we climbed hills or trees.
Learned of love and ABC's,
skinned our hearts and skinned our knees.
— Terry Jacks, *Seasons in the Sun*

timshel—'Thou Mayest'
— John Steinbeck, *East of Eden*

Schieb' den Wal zurück ins Meer!
— Die Toten Hosen, *Walkampf*

Danke

Dieser Abschnitt ist all denjenigen gewidmet, die ihr Scherflein zum Gelingen der vorliegenden Arbeit beigetragen haben.

Ich möchte zuallererst meinen Eltern danken, die mich während meiner gesamten Ausbildung in jedweder nichtwissenschaftlichen Weise gefördert haben.

Mein ganz besonderer Dank gilt auch Herrn Prof. Paul für die Vergabe des überaus spannenden und herausfordernden Themas und die wissenschaftliche Unterstützung meiner Promotion.

Ein dreifaches Hurrah und Prosit auch an das gesamte VAMP-Team für das Gelingen des Projekts.

Ein dickes Lob gebührt auch Werner Backes und meinen Freunden am Lehrstuhl, insbesondere Mark Hillebrand, Thomas In der Rieden, Christian Jacobi, Dirk Leinenbach und Jochen Preiß für das gute Lehrstuhl-Klima und zahlreiche fruchtbare Diskussionen sowie Bierbrauen, massierte Hähnchen, Doppelkopf-Abende und Kicker-Partien.

Und schließlich möchte ich auch Silvia Dorbach, Holger Herff und Christoph Schmidt herzlich danken für die unzähligen Abende in den letzten 15 Jahren, bei denen ich mich einfach vollständig zu Hause gefühlt habe.

Abstract

In this thesis we describe the formal verification of a cache memory interface and its integration into a microprocessor called VAMP. The cache memory interface and the VAMP are modeled on the gate level and verified against their respective specifications, i.e., a dual-ported memory for the cache memory interface and the programmer's model of the VAMP.

The cache memory interface features separate instruction and data caches with write back policy for the data cache; the caches are connected to a unified physical memory accesses via a bus protocol with bursts. The VAMP is an out-of-order 32 bit RISC CPU with DLX instruction set, fully IEEE-compliant floating point units, and a memory unit with a cache memory interface. The VAMP also supports precise interrupts. The formal verification of the out-of-order algorithm and the floating point units of the VAMP is not subject of this thesis; we 'only' put all the different parts together to an overall correctness proof.

Kurzzusammenfassung

In dieser Arbeit beschreiben wir die formale Verifikation eines Cache Memory Interfaces und dessen Integration in einen Mikroprozessor, den VAMP. Das Cache Memory Interface und der VAMP werden auf der Gatterebene modelliert und gegen ihre Spezifikation verifiziert, also einen Speicher mit zwei Zugriffsports für das Cache Memory Interface und das Programmiermodell des VAMP.

Das Cache Memory Interface besteht aus getrennten Instruktions- und Daten-Caches mit write-back Policy für den Daten-Cache. Die Caches sind mit einem vereinten physikalischen Speicher verbunden, auf den mittels eines Busprotokolls mit Bursts zugegriffen wird. Der VAMP ist eine out-of-order 32-bit RISC CPU mit DLX-Instruktionssatz, vollständig IEEE-konformen Fließkommaeinheiten und einer Speicher-Einheit mit einem Cache Memory Interface. Der VAMP unterstützt auch präzise Interrupts. Die formale Verifikation des out-of-order Algorithmus und der Fließkommaeinheiten des VAMP ist nicht Gegenstand dieser Arbeit; wir setzen lediglich die verschiedenen Teile zusammen zu einem Gesamt-Korrektheitsbeweis.

Extended Abstract

In this thesis we report on the formal verification of a cache memory interface and its integration into the out-of-order microprocessor VAMP [BJK+03, BJK+05]. Both the implementation of the circuits and their verification are carried out in the theorem proving system PVS [OSR92]. The design of the cache memory interface was inspired by the textbook on computer architecture by Müller and Paul [MP00] while its verification is completely new. Note that the design also extends the work of Müller and Paul by integrating write back policy for the data cache.

The VAMP is a pipelined out-of-order [Krö01] 32-bit RISC CPU with DLX instruction set, fully IEEE 754 [Ins85] compliant floating point units [Ber01, BJ01, Jac02a, Jac02b] for single- and double precision operations, a memory unit with a cache memory interface, delayed PC, and precise interrupts. The design and the formal verification of the Tomasulo [Tom67] out-of-order algorithm and the VAMP's floating point units and are based on the PhD-theses of Daniel Kröning [Krö01] and Christian Jacobi [Jac02a], respectively. We extend their work by actually verifying an implementation of the Tomasulo algorithm with the floating point units, adding instruction fetch and a memory unit, as well as by integrating precise interrupts into the proof of the VAMP.

The verification is split into three parts in a bottom-up fashion. As a first step, we develop a correctness criterion for simple caches called *cache consistency*. We take the cache implementations from [MP00] and formally verify these implementations to be consistent caches. In addition to the implementations proposed by [MP00], we give an implementation of a so-called fully associative cache and verify its consistency.

The second part of this thesis covers the formal verification of a cache memory interface. In particular, this cache memory interface consists of two caches for instruction fetch and data memory accesses, respectively. However, in this part, the caches are just black-boxed circuits fulfilling the above defined cache consistency. We define a correctness criterion for a cache memory interface. This criterion basically states that the cache memory interface behaves just like a dual-ported memory. We also describe the full implementation of the cache memory interface and its formal verification against its specification. In particular, since both caches are connected to a physical memory that is accessed with so-called bursts, this formal verification entails the full formalization of such a bus protocol which is completely new.

In the last step, we formally verify the overall correctness of the VAMP. For this level, the cache memory interface of the VAMP is just a black box with the correctness criterion verified in the previous step. We start with a definition of the programmer's model for the VAMP and a description of the VAMP implementation. The verification is then split into two sub-steps,

i.e., the correctness without interrupts and the correctness with interrupts. For the correctness without interrupts, we formally verify instruction fetch and the memory unit and instantiate the correctness of the Tomasulo algorithm by Kröning [Krö01]. In the last step, we add interrupts to the proof by decomposing arbitrary computations into interrupt-free parts, apply correctness without interrupts to these parts, and only investigate some additional proof into the cycles where an interrupts actually occurs.

All three steps together yield a formally verified gate-level implementation of the VAMP with interrupts and a cache memory interface with split instruction and data caches. The advantage of the different layers of abstraction corresponding to the above proof steps, however, is that is allows for concise and efficient reasoning, i.e., proofs are carried out only in the one small layer they logically belong to instead of the arguing over the huge overall implementation.

Together with the VAMP project team, we have also developed a translation tool [BJKL02] that takes our implementation of the VAMP from PVS and translates it to the hardware description language Verilog. This proves that real hardware can be synthesized from our implementation and we did not use any abstractions that cannot be represented by simple bits. The VAMP is currently running on a Xilinx FPGA on a PCI board at our institute.

Zusammenfassung

Die vorliegend Arbeit behandelt die formale Verifikation eines Cache Memory Interfaces und seine Integration in den out-of-order Mikroprozessor VAMP [BJK+03, BJK+05]. Sowohl die Implementierung der Schaltkreise als auch ihre Verifikation erfolgen im Beweis-System PVS [OSR92]. Das Design des Cache Memory Interface wurde vom Rechnerarchitektur-Lehrbuch von Müller und Paul [MP00] inspiriert; seine Verifikation dagegen ist vollständig neu. Auch das Design erweitert die Arbeit von Müller und Paul durch die Integration von write-back Policy für den Daten-Cache.

Der VAMP is eine gepipelinete out-of-order [Krö01] 32-bit RISC CPU mit einem DLX-Instruktionssatz, vollständig IEEE 754 [Ins85] konformen Fließkommaeinheiten [Ber01, BJ01, Jac02a, Jac02b] für einfache und doppelte Genauigkeit, einer Speicher-Einheit mit einem Cache Memory Interface, delayed PC und präzisen Interrupts. Das Design und die formale Verifikation des Tomasulo [Tom67] out-of-order Algorithmus und der Fließkommaeinheiten des VAMP basieren auf den Dissertationen von Daniel Kröning [Krö01] und Christian Jacobi [Jac02a]. Wir erweitern die Arbeit der beiden indem wir tatsächlich eine Implementierung des Tomasulo Algorithmus mit Fließkommaeinheiten, den Instruction Fetch, die Memory-Einheit und die Integration von präzisen Interrupts verifizieren.

Die Verifikation gliedert sich bottom-up in drei Teile. Als erstes entwickeln wir ein Korrektheitskriterium für einfache Caches; wir nennen es *Cache-Konsistenz*. Wir nehmen die Cache-Implementierungen aus [MP00] und verifizieren formal, dass diese Implementierungen konsistente Caches sind. Zusätzlich zu den Implementierungen, die in [MP00] vorgeschlagen werden, geben wir die Implementierung eines sogenannten voll-assoziativen Caches an und verifizieren dessen Konsistenz.

Der zweite Teil dieser Arbeit ist der formalen Verifikation eines Cache Memory Interfaces gewidmet. Ein Cache Memory Interface besteht insbesondere aus zwei Caches für Instruction Fetch und Zugriffe auf den Datenspeicher. In diesem Teil sind die Caches aber nur black-boxed Schaltkreise, welche die oben definierte Cache-Konsistenz erfüllen. Wir definieren zunächst ein Korrektheitskriterium für ein Cache Memory Interface. Dieses Kriterium behauptet im wesentlichen, dass sich das Cache Memory Interface wie ein Speicher mit zwei Zugriffsports verhält. Wir beschreiben auch die vollständige Implementierung des Cache Memory Interfaces und dessen formale Verifikation gegen seine Spezifikation. Da beide Caches mit einem physikalischen Speicher verbunden sind, auf den mit sogenannten Bursts zugegriffen wird, benötigt diese formale Verifikation insbesondere die vollständige Formalisierung eines solchen Busprotokolls, was gänzlich neu ist.

Im letzten Schritt verifizieren wir die vollständige Korrektheit des VAMP formal. Auf diesem Level ist das Cache Memory Interface des VAMP einfach

eine Black Box mit dem Korrektheitskriterium, dass wir im vorhergehenden Schritt verifiziert haben. Wir geben zunächst eine Definition für das Programmiermodell des VAMP und eine Beschreibung der Implementierung des VAMP an. Die Verifikation gliedert sich dann in zwei weitere Schritte, nämlich die Korrektheit ohne und mit Interrupts. Wir verifizieren den Instruction Fetch und die Speicher-Einheit formal und instantiieren den Korrektheitsbeweis des Tomasulo-Algorithmus von Kröning [Krö01] für die Korrektheit ohne Interrupts. Im letzten Schritt erweitern wir den Beweis um Interrupts, indem wir beliebige Berechnungen in Interrupt-freie Abschnitte unterteilen, für die wir Korrektheit ohne Interrupts anwenden, und nur noch die Korrektheit derjenigen Takte beweisen, in denen tatsächlich ein Interrupt auftritt.

Mit allen drei Schritten zusammen erhalten wir eine formal verifizierte Implementierung des VAMP mit Interrupts und einem Cache Memory Interface mit getrennten Instruktions- und Daten-Caches auf der Gatterebene. Der Vorteil der verschiedenen Abstraktionsebenen liegt aber darin, dass sie eine präzise und effiziente Argumentation erlauben, denn Beweise werden nur in der kleinen Abstraktionsebene geführt, in die sie logisch auch gehören, anstatt über die riesige komplette Implementierung zu argumentieren.

Zusammen mit dem VAMP Projekt Team haben wir auch ein Übersetzungstool [BJKL02] entwickelt, das unsere VAMP Implementierung in PVS in die Hardware-Beschreibungssprache Verilog übersetzt. Dadurch wird bewiesen, dass wir in der Implementierung keine Abstraktionen benutzt haben, die sich nicht in einfachen Bits repräsentieren lassen. Der VAMP läuft zur Zeit auf einem Xilinx FPGA auf einer PCI-Karte an unserem Lehrstuhl.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *'HAL, you have an enormous responsibility on this mission, in many ways perhaps the greatest responsibility of any single mission element. You're the brain, and central nervous system of the ship, and your responsibilities include watching over the men in hibernation. Does this ever cause you any lack of confidence?'*
>
> *'Let me put it this way, Mr. Amor. The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the word, foolproof and incapable of error.'*

This is a famous dialog from Stanley Kubrick's movie adaption [Kub68] of Arthur C. Clarke's '2001: A Space Odyssey' [Cla90]. HAL 9000 is a supercomputer who controls a human mission that aims at establishing contact with an alien species in the year 2001. Unfortunately, at some point, HAL decides to run amok and murder the human crew since, in his point of view, they hinder the success of the mission. However, a lone survivor of the crew succeeds in shutting down the higher functions of HAL's artificial intelligence and completes the mission alone.

HAL is obviously a highly complex computer system. Although no computer of his kind has ever made a mistake, this obviously does not guarantee absence of errors in the future. The 9000 series has been submitted to excessive testing, but due to HAL's complexity, simulation of all possible scenarios might just take several millennia. Hence, the only viable solution lies in formal verification. By formally verifying that HAL will under no circumstances consider his human crew 'expendable', one could have given the whole movie a new direction. This would have been very fortunate for the human crew, but on the other hand would rob us of a truly remarkable motion picture.

Seriously, however, there are many applications where microprocessors are employed in life-critical devices although these devices have not yet reached the complexity of HAL 9000. Consider, e.g., the medical sector,

nuclear power plants, cars, airplanes, or missiles. Due to the increasing complexity of these devices, the human effort invested in simulation increases exponentially, while simulation is less and less able to guarantee full security. Hence, formal verification more and more turns out to be the only alternative since the actual result of a formal verification is equivalent to a simulation of *all* possible cases, while it scales much better than simulation when putting the correctness of several modules together.

In this thesis, we consider the formal verification of a microprocessor called VAMP [BBJ+02, BJK+03, BJK+05] (Verified Architecture Microprocessor). Our particular focus lies on the cache memory interface of the VAMP which is based on the textbook of Müller and Paul [MP00, Chap. 6]. Our design extends their work by adding write back support to the data cache. All the proofs of the memory interface we present in this thesis are *new*.

The VAMP is a 32-bit RISC CPU with DLX instruction set. In addition to a memory unit with a cache memory interface, the VAMP features a Tomasulo [Tom67] out-of-order scheduler [Krö01], three floating point units [Ber01, BJ01, Jac02a, Jac02b], a fixed point unit, and precise interrupts. For the formal verification of the VAMP, we focus on the memory unit, instruction fetch, and precise interrupts in this thesis. We prove the overall VAMP implementation on the gate level correct with respect to a programmer's model that just executes one instruction in a step, i.e., we put all the different proofs together to one simple correctness statement. This correctness statement has also been formally verified in the theorem-proving system PVS [OSR92], i.e., all the proofs missing in this thesis due to lack of space have actually been carried out.

This thesis only deals with hardware verification. In order to formally verify the HAL 9000 system, however, both hardware and software correctness are needed. Hardware verification is just the first step and establishes a sound basis for software verification. In the discussion of future work, we will therefore give an outlook on the formal verification of software on top of verified hardware.

Outline

In the remaining chapter 1 we introduce the VAMP project, basic notation and basic circuits, the PVS system, and the general proof decomposition approach of this thesis.

Chapters 2 to 4 correspond to the three verification steps we presented in the extended abstract. Chapter 2 gives implementations and correctness proofs of parameterized caches. We present the design and verification of a cache memory interface in chapter 3. Finally, in chapter 4 we put all the different parts together to a correctness proof of the overall VAMP implementation.

Chapter 5 summarizes the results, discusses advantages and drawbacks of

our approach, and gives directions for future work. Related work is discussed at the end the respective chapters.

1.1 The VAMP project

The work presented here is part of our institute's *Verified Architecture Microprocessor* (VAMP) project [JK00, BBJ+02, BJK+03, BJK+05] where we specified an instruction set architecture, implemented a complete microprocessor on the gate level, and formally verified it in PVS [OSR92] against its specification given by the instruction set architecture. The VAMP is a pipelined 32-bit RISC microprocessor based on the MIPS instruction set featuring precise interrupts, a Tomasulo scheduler [Krö01], a fixed point unit, a pipelined, fully IEEE 754 [Ins85] compliant FPU, and a cache memory interface.

The floating point units [Ber01, BJ01, Jac02a, Jac02b] support addition/ subtraction, multiplication/ division, and test and convert operation. Division is computed by a Newton-Raphson iteration with finite precision which requires a loop in the pipeline structure of the floating point unit. Both single and double precision operations are supported by the hardware; additionally, denormal numbers are handled in hardware. The full set of exceptions required by the IEEE standard is supported.

The fixed point unit consists of an ALU and a shifter; it also supports test operations. The memory unit supports byte, halfword, word, and doubleword accesses of arbitrary, variable latency. In addition, load operations can be signed or unsigned. The cache memory interface supports write back for the data cache. In addition, instruction fetch is performed in the memory unit. Instruction and data cache are connected to a unified physical memory which is accessed via a bus protocol featuring bursts.

The VAMP supports maskable precise interrupts; return from the interrupt handler is achieved by means of an instruction. In addition, some interrupts re-execute the interrupted instruction while others continue with the next instruction. Typical supported interrupts are on illegal instruction or misaligned access, trap, and the interrupts provided by the floating point units as required by the IEEE standard.

Our hardware is written in a small subset of the PVS language. We use recursion and module instantiation for structured design, but the complete design can be unrolled down to the level of single bits and gates. This subset of the PVS language can be easily translated into common hardware description languages. For that purpose, we have developed a translation tool `pvs2hdl` [BJKL02] that takes our PVS hardware description and translates it into gate-level Verilog HDL. This translation process basically consists of unrolling recursive PVS implementations like adders into non-recursive Verilog modules and flattening nested record data structures employed in

PVS to multiple bitvectors in Verilog. The tool `pvs2hdl` is not verified.

Additionally, we ported the `gcc` and the `glibc` to the VAMP architecture [Mey02]; again, no verification was done. The VAMP is currently running on a Xilinx FPGA on an PCI board in our institute [Lei02]. We had to add some unverified circuits on the FPGA, e.g., in order to bridge the gap between SD-RAM and our bus protocol and to allow for host access to the SD-RAM. The Xilinx synthesis software used to create the FPGA configuration is also not verified. However, the verified VAMP is running on the PCI board and we did not find a single bug after completing the VAMP's formal verification in PVS. All PVS specifications and proofs, the Verilog files, and the sources of `pvs2hdl`, `gcc`, and `glibc` are available at our web site.[1]

1.2 Notation

In this section we introduce some shorthand notations, properties of binary numbers, and a few basic concepts for reasoning about input sequences. For the whole thesis, $\mathbb{N}$ denotes the natural numbers *including 0* and $\mathbb{N}^+ := \mathbb{N}\backslash\{0\}$; the set of integers, i.e., $\{\ldots,-1,0,1,\ldots\}$ is denoted by $\mathbb{Z}$. We start with some shorthand notation for sets.

Definition 1.2.1 *Let $n, m \in \mathbb{Z}$ be numbers. We define the following* ***integer intervals****:*

$$\begin{aligned}
[n:m] &:= [n,m] \cap \mathbb{Z} \\
]n:m] &:=]n,m] \cap \mathbb{Z} \\
[n:m[&:= [n,m[\cap \mathbb{Z} \\
]n:m[&:=]n,m[\cap \mathbb{Z} \\
\mathbb{Z}_n &:= [0:n[\\
\mathbb{Z}_{\geq n} &:= \mathbb{N} \setminus \mathbb{Z}_n
\end{aligned}$$

We now want to introduce bitvectors and arrays. We start with a standard definition of a word which is based on [LMW86].

Definition 1.2.2 *Let $\Sigma \neq \emptyset$ be a set called* ***alphabet****. A* ***word*** *of length $n \in \mathbb{N}$ over the alphabet Σ is a function $a : \mathbb{Z}_n \to \Sigma$. A word a is uniquely identified by the n-tuple of values $(a(n-1), a(n-2), \ldots, a(0))$. As a shorthand for this tuple, we also use $a_{n-1}a_{n-2}\ldots a_0$ or just $a[n-1:0]$. The set $\Sigma^n := \{a | a : \mathbb{Z}_n \to \Sigma\}$ is the set of all words of length n over Σ. We call $\epsilon := \Sigma^0$ the empty word. The set of all words over Σ is given by*

[1] http://www-wjp.cs.uni-sb.de/projects/verification/

$\Sigma^* := \bigcup_{n\in\mathbb{N}} \Sigma^n$. *The* ***concatenation*** *of words is defined by*

$$\begin{array}{rcl} \cdot : \Sigma^* \times \Sigma^* & \rightarrow & \Sigma^* \\ a[n-1:0] \cdot b[m-1:0] & = & (a(n-1), \ldots, a(0), b(m-1), \ldots b(0)) \end{array}$$

Instead of writing $\cdot$ *as infix operator, we also simple use* $a[n-1:0]b[m-1:0]$ *for concatenation.*

Definition 1.2.3 *A* ***domain*** *is an alphabet. An* ***array*** *is a word. We use the shorthand notation* $\mathbb{B} := \{0,1\}$. *A* ***bitvector*** *of length* n *is an array of length* n *over the domain* $\mathbb{B}$.

Note that the concept of arrays of length n over the domain D as words actually corresponds to the informal standard idea of arrays as containing n values over the domain D that are addressed by $a[i]$ for $i \in \mathbb{Z}_n$. In addition, we can naturally have multi-dimensional arrays, e.g., by using $\mathbb{B}^n$ as a domain of an array. Note also that we will not use $\cdot$ for the concatenation of arbitrary arrays, but only for bitvectors since array concatenation is not a common operation.

By defining arrays as functions, we can also use λ-notation for arrays since λ-notation introduces unnamed functions. Note that we *only* use λ-notation as purely syntactical sugar for shorter notations, i.e., we do not employ real λ-calculus. Consider, e.g., a an array a of length m over the domain $\mathbb{B}^n$. A function $upd(a, i, val)$ that updates index i of array a by val and leaves all other values untouched is defined by

$$upd(a, i, val)[j] := (j = i)?\, val : a[j],\ (j \in \mathbb{Z}_m).$$

Instead, with λ-notation, we only have a single equality instead of one equality per array index.

$$upd(a, i, val) := \lambda_{j\in\mathbb{Z}_m}(j = i)?\, val : a[j]$$

Definition 1.2.4 *Let* $n \in \mathbb{N}^+$ *and* $a \in \mathbb{B}^n$. *We call*

$$\langle a \rangle := \sum_{i=0}^{n-1} a[i] \cdot 2^i$$

the ***binary number*** *represented by* a. *Note that* $\langle\cdot\rangle : \mathbb{B}^n \rightarrow \mathbb{Z}_{2^n}$ *is bijective.*[2] *Thus, the function* $\mathrm{bin}_n := \langle\cdot\rangle^{-1}, \mathrm{bin}_n : \mathbb{Z}_{2^n} \rightarrow \mathbb{B}^n$ *that returns the* ***binary representation*** *of a natural number is well founded.*

Additionally, we identify the value 1 of a bit with the boolean value *true* and 0 with *false* for this thesis. This avoids tedious conversions on the bit level that are necessary in PVS.

[2] This property is formally verified in the PVS standard library `bitvectors`, cf. section 1.3.

Proposition 1.2.5 *Let $a, b \in \mathbb{B}^n$, $m \in \mathbb{Z}_n$, and $k \in \mathbb{Z}_{2^m}$. The following statements hold:*

- $\langle a \rangle \in \mathbb{Z}_{2^n}$
- $\langle a \rangle = \langle b \rangle \iff a = b$
- $\langle a \rangle = 2^m \cdot \langle a[n-1:m] \rangle + \langle a[m-1:0] \rangle$
- $\langle a \rangle \bmod 2^m = k \iff \langle a[m-1:0] \rangle = k$

Proofs for these simple properties are available in the PVS `bitvectors` library we introduce in section 1.3 and are therefore omitted in this thesis. In our hardware implementations, we often use RAM in contrast to arrays. Therefore, we introduce the necessary notations in this section.

Definition 1.2.6 *For any $a \in \mathbb{N}$ and $d \in \mathbb{N}$, a $\mathbf{2^a \times d}$-**RAM** R is a function $\mathbb{B}^a \to \mathbb{B}^d$ that maps any input address $adr \in \mathbb{B}^a$ to its data value in R, denoted by $R[adr]$.*

Note that the only difference between an array and a RAM according to the above definition is in the type of the index variable, i.e., an array is always indexed by a natural number, whereas a RAM is indexed by a bitvector. Note that since a RAM is a function, we can also employ λ-notation for RAM. We now introduce some notation for the values of registers and signals in some cycle t.

Definition 1.2.7 *Let D_I a domain that does* not *contain RAM. A function $inp : \mathbb{N} \to D_I$ is then called an **input signal** over the domain D_I. We use the shorthand notation inp^t for the value of inp in cycle t, i.e., $inp(t)$.*

Let c_{init} be some initial configuration over the domain D_c, inp some input signal over the domain D_I, and $next_c : D_c \times D_I \to D_c$ a function that computes the next state of configuration c based on the current state and some input. We then denote the content of configuration c in cycle t given a starting configuration c_{init} with $c[c_{init}]^t$, i.e., we have

$$\begin{aligned} c[c_{init}]^0 &:= c_{init} \\ c[c_{init}]^{t+1} &:= next_c(c[c_{init}]^t, inp^t) \end{aligned}$$

If we do not explicitly care for the starting configuration, but just assume an arbitrary, but fixed one, we also simply write c^t.

*Let D_c and D_I be a configuration domain and an input signal domain, respectively. An **output signal** over the domain D_O is a function $out : D_c \times D_I \to D_O$ and we once again denote the value of an output signal out in cycle t with out^t, i.e., we have $out^t := out(c^t, inp^t)$. Note that we also write $out[c_{init}]^t$ if we refer to a special initial configuration c_{init}.*

In addition, we sometimes want to argue about some 'virtual' outputs, i.e., outputs given some hypothetical input $in \in D_I$ instead of the concrete input inp^t. Therefore, we introduce the notation $out^t_{in} := out(c^t, in)$. Note that $out^t_{inp^t} = out^t$ trivially holds.

Thus, we can now argue about signals and configurations in cycles and output signals based on hypothetical inputs. When arguing about the value of a signal s in different cycles, one often talks about the last time s was asserted or the next time s will be asserted. We now formalize these concepts and show a few simple properties. The corresponding PVS definitions and lemmas that we developed can be found in the context `predicates`.

Definition 1.2.8 *Let S be a signal over the domain D, P a predicate on D, and $t \in \mathbb{N}$ a cycle. We introduce the shorthand notation P^t for $P(S^t)$ and define a predicate indicating that P held in a cycle prior to t, i.e., $\exists^{last}_P(t) := \exists t' < t : P^{t'}$, and a second predicate indicating that P holds in the present cycle or in a future cycle, $\exists^{next}_P(t) := \exists t' \geq t : P^{t'}$.*

In case $\exists^{last}_P(t)$ holds, we define the last cycle where P held, i.e., $last_P(t) := \max\{t' < t : P^{t'}\}$. If $\exists^{next}_P(t)$ holds, we define the next cycle where P holds as $next_P(t) := \min\{t' \geq t : P^{t'}\}$.

If it is clear from the context which predicate P is considered, we will abbreviate $last_P(t)$ with $last(t)$ and $next_P(t)$ with $next(t)$.

The following proposition collects a few trivial properties of the *next* and *last* definitions. The proofs are omitted in this thesis since they are very easy and very similar.

Proposition 1.2.9 *Let S, D, P, and t be like in Definition 1.2.8. Then the following properties hold:*

- $\exists^{last}_P(t) \implies P^{last_P(t)} \wedge \forall t' \in]last_P(t) : t[: \neg P^{t'}$
- $\exists^{next}_P(t) \implies P^{next_P(t)} \wedge \forall t' \in [t : next_P(t)[: \neg P^{t'}$
- $P^0 \wedge t > 0 \implies \exists^{last}_P(t)$
- $t' \geq t \wedge \exists^{last}_P(t) \implies \exists^{last}_P(t')$
- $t' \leq t \wedge \exists^{next}_P(t) \implies \exists^{next}_P(t')$
- $t' \leq t \wedge P^{t'} \implies \exists^{last}_P(t) \wedge last_P(t) \geq t'$
- $t' \geq t \wedge P^{t'} \implies \exists^{next}_P(t) \wedge next_P(t) \leq t'$
- $t' \geq t \wedge \exists^{last}_P(t) \implies last_P(t') \geq last_P(t)$
- $t' \leq t \wedge \exists^{next}_P(t) \implies next_P(t') \leq next_P(t)$
- $t' \in]last_P(t) : t] \implies last_P(t) = last_P(t')$

As an example for the use of the definition of *last*, we introduce the following simple lemma about register values which we will use often when arguing about the value in some register or RAM.

Lemma 1.2.10 *Let R_{init} be an initial configuration over the domain D, R_{ce} a signal over the domain $\mathbb{B}$, and R_{in} a signal over the domain D. We then define the content of register R in cycle $t \in \mathbb{N}$ as follows:*

$$\begin{aligned} R^0 &:= R_{init} \\ R^{t+1} &:= \begin{cases} R_{in}^t & \text{if } R_{ce}^t \\ R^t & \text{otherwise} \end{cases} \end{aligned}$$

For any cycle $t \in \mathbb{N}$, it holds that

$$R^t = \begin{cases} R_{in}^{last_{R_{ce}}(t)} & \text{if } \exists_{R_{ce}}^{last}(t) \\ R_{init} & \text{otherwise} \end{cases}$$

where we identify the boolean signal R_{ce} with a predicate that holds iff R_{ce} is asserted.

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 0}$): By definition, we have $R^0 = R_{init}$ which concludes the claim since $\exists_{R_{ce}}^{last}(0)$ never holds.
Induction step ($\mathbf{t \to t + 1}$): By definition, we have

$$R^{t+1} := \begin{cases} R_{in}^t & \text{if } R_{ce}^t \\ R^t & \text{otherwise} \end{cases}$$

We now split cases on the value of R_{ce}^t.

1. If R_{ce}^t holds, we have $R^{t+1} = R_{in}^t$. In this case, $\exists_{R_{ce}}^{last}(t+1)$ trivially holds and also $last_{R_{ce}}(t+1) = t$ which concludes this case of the claim.

2. If R_{ce}^t does not hold, we trivially have $R^{t+1} = R^t$ and $\exists_{R_{ce}}^{last}(t+1) = \exists_{R_{ce}}^{last}(t)$. With the induction hypothesis for R^t, this yields

$$R^{t+1} = \begin{cases} R_{in}^{last_{R_{ce}}(t)} & \text{if } \exists_{R_{ce}}^{last}(t+1) \\ R_{init} & \text{otherwise} \end{cases}$$

Note that in case of $\neg\exists_{R_{ce}}^{last}(t+1)$, we have $R^{t+1} = R_{init}$ which concludes the proof. Let therefore $\exists_{R_{ce}}^{last}(t+1)$ hold. We then have $last_{R_{ce}}(t+1) = last_{R_{ce}}(t)$. Thus, we get $R^{t+1} = R^{last_{R_{ce}}(t+1)}$ which finishes the proof. □

1.3 The PVS system

The *Prototype Verification System* [OSR92], abbreviated PVS, is an interactive theorem prover developed at SRI International. PVS features powerful decision procedures for natural numbers and a `bitvectors` library [BMSG96] that is the basis for our hardware implementations. Basic PVS bitvector types are `bit` which equals $\mathbb{B}$ as defined in the previous section and `bvec[n]` for $\mathbb{B}^n$. Concatenation and bit extraction operators are defined in analogy to the concepts introduced in the previous section. In addition, the binary number represented by a bitvector is computed by a function `bv2nat` and the inverse is given by `nat2bv`. The library contains several simple lemmas about bitvectors which we just refer to in this thesis without giving a proof transcript like proposition 1.2.5.

Since bitvectors and arrays are also functions in PVS, the usage of λ-expressions in modelling bitvectors and arrays is natural. Note that PVS actually has to distinguish between $\mathbb{B}$ and $\mathbb{B}^1$ which we will not do in this thesis for the sake of readability.

The modeling of combinational hardware is straightforward in PVS, i.e., we employ functions which are possibly recursive and λ-expressions on the bitvector, array, or RAM types. You can basically take the design of the encoder in section 1.4.3 and just copy it to PVS while replacing $\mathbb{B}^n$ with `bvec[n]` and get the actual PVS implementation.[3] The same basically holds true for the specification; you only have to replace the functions bin and log by their PVS-counterparts.

Modeling of clocked circuits, however, is not as straightforward as for combinational circuits. This stems from the fact that we use a *functional* subset of the PVS language for hardware modeling and therefore, we have no global state-holding variables which would correspond to registers. Hence, we just model a next-state function that maps the current configuration and some input to the successor configuration and, potentially, some output. The state and output of the system in cycle t given an input sequence up to cycle t can then be defined recursively based on the next state function just as we introduced in definition 1.2.7.

The interactive theorem prover PVS features all the standard techniques used in paper-and-pencil proofs, i.e., skolemization, case distinctions, induction, application of lemmas, and instantiation of suitably quantified formulas, while it unfortunately lacks an 'obvious' button that solves the numerous subgoals we will just declare to be 'obvious' in this thesis. However, since we actually completed the formal verification in PVS, we *know* that all the subgoal we declare as 'obvious' in this thesis really are obvious, although some effort might have been necessary in order to convince PVS of that fact.

[3] Note that in this translation step, you have to take care of the distinction of $\mathbb{B}$ and $\mathbb{B}^1$ by PVS, e.g., you may have to add a λ-expression in order to convert a single bit into a bitvector.

On the other hand, in the one case in several dozens or so when PVS really was right and a subgoal was not 'obvious', we will present the details that defied obviousness in this thesis.

In summary, it is basically possible to translate any paper-and-pencil proof presented in this thesis into a formal PVS proof while keeping the proof structure we presented. Note that most of the proofs for this thesis have been developed directly in PVS and we just present a sanitized paper-and-pencil version of these proofs.

1.4 Basic circuits

In this section, we verify the correctness of various basic circuits that we will use as macros later on. We have already reported the verification of decoders, adders, equality-tester, and similar circuits in [BJK01]. Since we just use these macros and their proofs, we will not go into detail here apart from giving definitions. Instead, we will focus on additional macros like encoders and multiplexer trees.

Definition 1.4.1 *Let $n \in \mathbb{N}^+$. We then define the following parametrized circuits:*

- *An* **n-*ortree*** *is a circuit computing the function*

$$or_n : \mathbb{B}^n \to \mathbb{B},\ or_n(a) := \exists i \in \mathbb{Z}_n : a[i].$$

- *An* **n-*equality-tester*** *is a circuit computing the function*

$$eq_n : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B},\ eq_n(a,b) := (a = b).$$

- *An* **n-*decoder*** *is a circuit computing the function*

$$dec_n : \mathbb{B}^n \to \mathbb{B}^{2^n},\ dec_n(a) := \lambda_{i \in \mathbb{Z}_{2^n}}(\langle a \rangle = i).$$

- *An* **n-*adder*** *is a circuit computing the function*

$$add_n : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B}^{n+1},\ add_n(a,b,c) := \mathrm{bin}_{n+1}(\langle a \rangle + \langle b \rangle + \langle c \rangle).$$

- *An* **n-*incrementer*** *is a circuit computing the function*

$$inc_n : \mathbb{B}^n \to \mathbb{B}^{n+1},\ inc_n(a) := \mathrm{bin}_{n+1}(\langle a \rangle + 1).$$

We will just use the above specifications in all the proofs where the corresponding macros appear in our design. However, an implementation of, i.e., a decoder or an equality-tester together with a correctness proof is available in our PVS tree.

1.4.1 Multiplexer trees

Definition 1.4.2 *For $n \in \mathbb{Z}_{\geq 2}, m \in \mathbb{N}^+$, an* **n-*muxtree*** *is a circuit computing the function*

$$mux_n : (\mathbb{B}^m)^n \times \mathbb{B}^{\lceil \log n \rceil} \to \mathbb{B}^m,\ mux_n(A, b) := A[\langle b \rangle]$$

for any $b \in \mathbb{B}^{\lceil \log n \rceil}$ with $\langle b \rangle < n$.

For $n \in \mathbb{N}^+, m \in \mathbb{N}^+$, an **n-*muxtree with unary select*** *is a circuit computing a function $mux\_us_n : (\mathbb{B}^m)^n \times \mathbb{B}^n \to \mathbb{B}^m$ that fulfills for any Array A of size n over the Domain $\mathbb{B}^m$, $b \in \mathbb{B}^n$, and any $j \in \mathbb{Z}_n$*

$$(b = \lambda_{i \in \mathbb{Z}_n}(j = i)) \implies mux\_us_n(A, b) = A[j].$$

Lemma 1.4.3 *For $n \in \mathbb{N}, m \in \mathbb{N}^+$, the circuit $MUX_{2^n} : (\mathbb{B}^m)^{2^n} \times \mathbb{B}^n \to \mathbb{B}^m$ defined by*

$$\begin{aligned} MUX_{2^0}(A, b) &:= b[0]?\, A[1] : A[0] \\ MUX_{2^{n+1}}(A, b) &:= b[n]?\, MUX_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}} A[2^n + i], b[n-1:0]) : \\ &\quad\ MUX_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}} A[i], b[n-1:0]) \end{aligned}$$

is a 2^n-muxtree according to definition 1.4.2.

Proof: We show the claim $MUX_{2^n}(A, b) = A[\langle b \rangle]$ by induction on n.
Induction base ($\mathbf{n = 0}$): We have $MUX_{2^0}(A, b) = b[0]?A[1] : A[0]$ which obviously equals $A[\langle b \rangle]$.
Induction step ($\mathbf{n \to n + 1}$): Let $MUX_{2^n}(A', b') = A'[\langle b' \rangle]$ for all A', b' according to induction hypothesis. We then have by definition

$$\begin{aligned} MUX_{2^{n+1}}(A, b) = \ & b[n]?\, MUX_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}} A[2^n + i], b[n-1:0]) : \\ & MUX_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}} A[i], b[n-1:0]) \end{aligned}$$

By applying the induction hypothesis twice, we get

$$MUX_{2^{n+1}}(A, b) = b[n]?\, A[2^n + \langle b[n-1:0] \rangle] : A[\langle b[n-1:0] \rangle]$$

With Proposition 1.2.5, this easily leads to

$$MUX_{2^{n+1}}(A, b) = A[\langle b \rangle]$$

which concludes the claim. □

Lemma 1.4.4 *For $n \in \mathbb{N}^+, m \in \mathbb{N}^+$, the circuit $MUX\_us_n : (\mathbb{B}^m)^n \times \mathbb{B}^n \to \mathbb{B}^m$ defined by*

$$MUX\_us_n(A, b) := \lambda_{i \in \mathbb{Z}_m} or_n(\lambda_{j \in \mathbb{Z}_n} A[j][i] \wedge b[j])$$

is an n-muxtree with unary select according to definition 1.4.2.

Proof: Let $n \in \mathbb{N}+$, $m \in \mathbb{N}+$, $k \in \mathbb{Z}_n$ and $b := \lambda_{i\in\mathbb{Z}_n}(k = i)$. We then have to show $MUX\_us_n(A, b) = A[k]$. We start with the definition from lemma 1.4.3, i.e.,

$$MUX\_us_n(A, b) = \lambda_{i\in\mathbb{Z}_m} or_n(\lambda_{j\in\mathbb{Z}_{2^n}} A[j][i] \wedge b[j]).$$

By definition 1.4.1 for or_n we have

$$MUX\_us_n(A, b) = \lambda_{i\in\mathbb{Z}_m} \exists j \in \mathbb{Z}_n : A[j][i] \wedge b[j].$$

By replacing $b = \lambda_{j\in\mathbb{Z}_n}(k = j)$, we simply get

$$MUX\_us_n(A, b) = \lambda_{i\in\mathbb{Z}_m} A[k][i].$$

This concludes the claim since $A[k] = \lambda_{i\in\mathbb{Z}_m} A[k][i]$. □

1.4.2 Parallel prefix or

Definition 1.4.5 *For $n \in \mathbb{N}^+$, an* **n-*parallel prefix or*** *is a circuit computing the function $pp_n : \mathbb{B}^n \to \mathbb{B}^n$,*

$$pp_n(a) := \lambda_{i\in\mathbb{Z}_n} \bigvee_{j=0}^{j\le i} a[j].$$

Lemma 1.4.6 *For $n \in \mathbb{N}^+$, we set $m := \lfloor n/2 \rfloor$, $a' := \lambda_{l\in\mathbb{Z}_m} a[2\cdot l+1] \vee a[2\cdot l]$, and define a circuit PP_n recursively by*

$$\begin{aligned} PP_1(a) &:= a \\ PP_n(a) &:= \lambda_{l\in\mathbb{Z}_n} \begin{cases} a[0] & l = 0 \\ PP_m(a')[(l-1)/2] & l > 0 \wedge odd?(l) \\ PP_m(a')[l/2-1] \vee a[l] & l > 0 \wedge even?(l) \end{cases} \end{aligned}$$

Note that we use the predicates odd? and even? for odd and even integers, respectively. The circuit PP_n thus defined is a parallel prefix or according to definition 1.4.5.

Proof: We prove the claim $PP_n(a) = \lambda_{i\in\mathbb{Z}_n}(\bigvee_{j=0}^{j\le i} a[j])$ by induction on n.
Induction base ($\mathbf{n = 1}$): For $n = 1$, we have $PP_1(a) = a = \bigvee_{j=0}^{j\le 0} a[j]$ which concludes the induction base.
Induction step ($\mathbf{n \to n+1}$): We set $m := \lfloor (n+1)/2 \rfloor$ and additionally $a' := \lambda_{l\in\mathbb{Z}_m}(a[2\cdot l+1] \vee a[2\cdot l])$. Note that the induction hypothesis guarantees $PP_m(a') = \lambda_{i\in\mathbb{Z}_m}(\bigvee_{j=0}^{j\le i} a'[j])$. We then have by definition

$$PP_{n+1}(a) = \lambda_{l\in\mathbb{Z}_n} \begin{cases} a[0] & l = 0 \\ PP_m(a')[(l-1)/2] & l > 0 \wedge odd?(l) \\ PP_m(a')[l/2-1] \vee a[j] & l > 0 \wedge even?(l) \end{cases}$$

Applying the induction hypothesis yields

$$PP_{n+1}(a) = \lambda_{l\in\mathbb{Z}_n} \begin{cases} a[0] & l = 0 \\ \bigvee_{j=0}^{j\leq(l-1)/2} a'[j] & l > 0 \wedge odd?(l) \\ \bigvee_{j=0}^{j\leq l/2-1} a'[j] \vee a[l] & l > 0 \wedge even?(l) \end{cases}$$

Expanding the definition of a', we get

$$PP_{n+1}(a) = \lambda_{l\in\mathbb{Z}_n} \begin{cases} a[0] & l = 0 \\ \bigvee_{j=0}^{j\leq(l-1)/2} (a[2\cdot j+1] \vee a[2\cdot j]) & l > 0 \wedge odd?(l) \\ \bigvee_{j=0}^{j\leq l/2-1} (a[2\cdot j+1] \vee a[2\cdot j]) \vee a[l] & l > 0 \wedge even?(l) \end{cases}$$

In all three cases, this equals the desired result, i.e.,

$$PP_{n+1}(a) = \lambda_{l\in\mathbb{Z}_n} (\bigvee_{j=0}^{j\leq l} a[j])$$

which concludes the induction step. □

1.4.3 Encoder

Definition 1.4.7 *For $n \in \mathbb{Z}_{\geq 2}$, an* **n-*encoder*** *is a circuit computing a function $enc_n : \mathbb{B}^n \to \mathbb{B}^{\lceil \log n \rceil}$ that fulfills for any $a \in \mathbb{B}^n$ and $j \in \mathbb{Z}_n$*

$$(a = \lambda_{i\in\mathbb{Z}_n}(j = i)) \implies enc_n(a) = \mathrm{bin}_{\lceil \log n \rceil}(j)$$

Lemma 1.4.8 *For $n \in \mathbb{N}^+$, we define a circuit $encf_{2^n} : \mathbb{B}^{2^n} \to \mathbb{B}^{n+1}$ recursively by*

$$\begin{aligned} encf_{2^1}(a) &:= (a[1] \vee a[0]) \cdot a[1] \\ encf_{2^{n+1}}(a) &:= (lo[n] \vee hi[n]) \cdot \lambda_{i\in\mathbb{Z}_{n+1}} \begin{cases} hi[n] & i = n \\ lo[i] \vee hi[i] & otherwise \end{cases} \\ &\quad with\ lo := encf_{2^n}(a[2^n - 1 : 0]) \\ &\quad and\ hi := encf_{2^n}(a[2^{n+1} - 1 : 2^n]) \end{aligned}$$

This circuit is an extended encoder, i.e., $encf_{2^n}(a)[n-1:0] = enc_{2^n}(a)$ and $encf_{2^n}(a)[n] = or_{2^n}(a)$.

The implementation in the above lemma is taken from [MP95]. In order to verify this lemma, we first show the following two propositions.

Proposition 1.4.9 *For $n \in \mathbb{N}^+$ and $a \in \mathbb{B}^n$, the highest bit of $encf_{2^n}$ computes just an n-ortree, i.e., $encf_n(a)[n] = or_n(a)$.*

Since the recursive definition of $encf_{2^n}(a)[n]$ exactly equals the definition of an *or*-tree, this proposition is trivially verified.

Proposition 1.4.10 *For $n \in \mathbb{N}^+$, it holds that $encf_{2^n}(0^{2^n}) = 0^{n+1}$.*

Proof: One part of this proposition, i.e., $encf_{2^n}(0^{2^n})[n] = 0$ is already given by proposition 1.4.9. For the remaining bits, we show the claim $encf_{2^n}(0^{2^n})[n-1:0] = 0^n$ by induction on n.
Induction base ($\mathbf{n = 1}$): We have by definition $encf_{2^1}(00)[0] = 0$ which concludes the induction base.
Induction step ($\mathbf{n \to n+1}$): Let $encf_{2^n}(0^{2^n})[n-1:0] = 0^n$ hold by induction hypothesis. By definition, we have

$$encf_{2^{n+1}}(0^{2^{n+1}})[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} encf_{2^n}(0^{2^n})[n] & i = n \\ encf_{2^n}(0^{2^n})[i] \vee encf_{2^n}(0^{2^n})[i] & \text{otherwise} \end{cases}$$

We conclude the claim with the induction hypothesis and with proposition 1.4.9. □

With these two propositions, we are now finally able to prove lemma 1.4.8 correct.
Proof: (of Lemma 1.4.8) Let $n \in \mathbb{N}^+$, $j \in \mathbb{Z}_{2^n}$, and $a := \lambda_{i \in \mathbb{Z}_{2^n}}(j = i)$. Note that $\lceil \log 2^n \rceil = n$ trivially holds. We show $encf_{2^n}(a)[n-1:0] = \mathrm{bin}_n(j)$ by induction on n.
Induction base ($\mathbf{n = 1}$): We have two cases depending on the value of j.

1. For $j = 0$, we have $encf_{2^1}(01)[0] = a[1] = 0 = \mathrm{bin}_1(0)$ which concludes the claim.

2. For $j = 1$, we have $encf_{2^1}(10)[0] = a[1] = 1 = \mathrm{bin}_1(1)$ which also concludes the claim.

Thus the claim holds for the induction base.
Induction step ($\mathbf{n \to n+1}$): Let $j \in \mathbb{Z}_{2^{n+1}}$, $a := \lambda_{l \in \mathbb{Z}_{2^{n+1}}}(j = l)$, and $encf_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}}(j' = i))[n-1:0] = \mathrm{bin}_n(j')$ hold by induction hypothesis for any $j' \in \mathbb{Z}_{2^n}$. We then have by definition

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} hi[n] & i = n \\ lo[i] \vee hi[i] & \text{otherwise} \end{cases}$$

with $lo := encf_{2^n}(a[2^n - 1 : 0])$ and $hi := encf_{2^n}(a[2^{n+1} - 1 : 2^n])$ just as in lemma 1.4.8. By replacing hi, lo, and a on the right hand side, we get

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} encf_{2^n}(\lambda_{l \in \mathbb{Z}_{2^n}}(j = l + 2^n))[n] & i = n \\ encf_{2^n}(\lambda_{l \in \mathbb{Z}_{2^n}}(j = l))[i] \vee & \text{otherwise} \\ \quad encf_{2^n}(\lambda_{i \in \mathbb{Z}_{2^n}}(j = l + 2^n))[i] & \end{cases}$$

Depending on the value of j, we now distinguish two cases.

1. For $j \in \mathbb{Z}_{2^n}$, we have $\lambda_{l \in \mathbb{Z}_{2^n}}(j = l + 2^n) = 0^{2^n}$ and thus, we can apply proposition 1.4.10 in order to get

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} 0 & i = n \\ encf_{2^n}(\lambda_{l \in \mathbb{Z}_{2^n}}(j = l))[i] & \text{otherwise} \end{cases}$$

We now define $j' := j$ and apply the induction hypothesis. This yields

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} 0 & i = n \\ \mathrm{bin}_n(j')[i] & \text{otherwise} \end{cases}$$

Since $j' \in \mathbb{Z}_{2^n}$, we have $\mathrm{bin}_{n+1}(j')[n] = 0$, and thus the claim is concluded by $encf_{2^{n+1}}(a)[n:0] = 0 \cdot \mathrm{bin}_n(j') = \mathrm{bin}_{n+1}(j') = \mathrm{bin}_{n+1}(j)$.

2. For $j \geq 2^n$, the arguments are similar. We have $\lambda_{l \in \mathbb{Z}_{2^n}}(j = l) = 0^{2^n}$, and apply propositions 1.4.10 and 1.4.9 and definition 1.4.1 in order to get

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} \exists l \in \mathbb{Z}_{2^n} : j = l + 2^n & i = n \\ encf_{2^n}(\lambda_{l \in \mathbb{Z}_{2^n}}(j = l + 2^n))[i] & \text{otherwise} \end{cases}$$

Since $j \in [2^n : 2^{n+1}[$, we define $\mathbb{Z}_{2^n} \ni j' := j - 2^n$ and thus, we have $(\exists l \in \mathbb{Z}_{2^n} : (j = l + 2^n)) = 1$. We apply the induction hypothesis in order to get

$$encf_{2^{n+1}}(a)[n:0] = \lambda_{i \in \mathbb{Z}_{n+1}} \begin{cases} 1 & i = n \\ \mathrm{bin}_n(j')[i] & \text{otherwise} \end{cases}$$

Since $j \in [2^n : 2^{n+1}[$, we have $\mathrm{bin}_{n+1}(j)[n] = 1$, and thus the claim is concluded by $encf_{2^{n+1}}(a)[n:0] = 1 \cdot \mathrm{bin}_n(j') = \mathrm{bin}_{n+1}(j)$. □

We now trivially extend the encoder to input widths that are not a power of 2.

Definition 1.4.11 *For $n \in \mathbb{Z}_{\geq 2}$, we set $k := \lceil \log n \rceil$. We extend the definition of encf from lemma 1.4.8 by*

$$encf_n(a) := encf_{2^k}(0^{2^k - n} \cdot a).$$

This extended *encf* circuit trivially fulfills the correctness criteria presented in lemma 1.4.8, i.e., if $a = \lambda_{i \in \mathbb{Z}_n}(i = j)$ for some $j \in \mathbb{Z}_n$, we have

$$\begin{aligned} encf_n(a)[k] &= or_n(a) \\ encf_n(a)[k-1:0] &= \mathrm{bin}_k(j) \end{aligned}$$

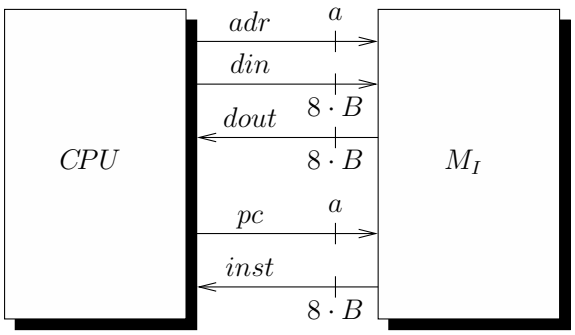

Figure 1.1: Data paths of a memory interface

1.5 Proof decomposition

We now want to give an overview on how the overall correctness proof of the VAMP with its cache memory interface is structured hierarchically. This proof decomposition allows for concise arguments while abstracting away all the information that is not needed for a proof. In particular, we will introduce the layers of a memory interface and a consistent cache in the following sections.

For the remaining chapter, let $B \geq 1$ be the number of bytes stored in a single memory location, and let $a \geq 1$ be the number of bits used in addressing the memory. Thus, a *data word* w consists of B bytes, i.e., $w \in \mathbb{B}^{8 \cdot B}$, and our memory contains $2^a \cdot B$ bytes. As a first step, we introduce the memory interface layer. The overall correctness proof of the VAMP will exclusively use this layer for any memory access.

1.5.1 The memory interface layer

Definition 1.5.1 *A **memory interface** for a pipelined microprocessor is a circuit with inputs and outputs according to table 1.1; its data paths are depicted in figure 1.1.*

*We call the CPU output to the memory interface **valid** if there is only an initial clear, any data or instruction access is stalled by an active dbusy or ibusy, respectively, and the read- and write signals on the data port are never raised simultaneously. Formally, we have*

- $\forall t \in \mathbb{N} : clear^t = (t = 0)$
- $\forall t \in \mathbb{N}^+ : \neg mw^t \vee \neg mr^t$
- $\forall t \in \mathbb{N}^+ : (mr^t \vee mw^t) \wedge dbusy^t \implies \{adr, din, mw, mr, mwb\}^{t+1} = \{adr, din, mw, mr, mwb\}^t$
- $\forall t \in \mathbb{N}^+ : imr^t \wedge ibusy^t \implies \{pc, imr\}^{t+1} = \{pc, imr\}^t$

Signal	Description
Memory interface input	
$adr[a-1:0]$	word address of the data access
$din[8 \cdot B - 1 : 0]$	data word to be written
mw	signals data write access
mr	signals data read access
$mwb[B-1:0]$	selects bytes of the data word to be written
$pc[a-1:0]$	word address of the instruction access
imr	signals instruction read access
$clear$	initializes memory interface
Memory interface output	
$dbusy$	signals pending data access
$dout[8 \cdot B - 1 : 0]$	read data on finished data access
$ibusy$	signals pending instruction access
$inst$	read data on finished instruction access

Table 1.1: Interface between a CPU and a memory interface

The timing in our memory interface is simple. The CPU starts a data request in a cycle t by raising mr^t for a read or mw^t for a write. The address of the request is adr^t, and in case of a write, din^t holds the data to be written and mwb^t contains the byte enables for the single bytes in din^t. All these signals keep their value until in a cycle $t' \geq t$, $dbusy^{t'}$ is lowered. In case of a read access, the data returned on $dout^{t'}$ is the requested data.

Similarly, an instruction read request is started in a cycle t by an active mr^t. The address pc^t remains stable until in a cycle $t' \geq t$, $ibusy^{t'}$ is lowered. The data output $inst^{t'}$ is the requested instruction data. Instruction and data request may be arbitrarily interleaved. This is illustrated in figure 1.2.

We call a memory interface with valid input from a CPU *correct* if it is both live and consistent. Liveness means that any access to the memory interface eventually terminates, and consistency means that any read access yields the expected data. The following definition formalizes these concepts.

Definition 1.5.2 *Let* $init\_mem \in (\mathbb{B}^{8 \cdot B})^{2^a}$ *be the initial memory content of a memory interface. For any* $w \in \mathbb{B}^{8 \cdot B}$ *and* $b < B$*, we use the shorthand notation* $|w|_b = w[8 \cdot b + 7 : 8 \cdot b]$ *for the projection of word* w *to its* b*-th byte.*

We introduce a parameterized predicate on the memory interface I/O by

$$M_I.bw(ad, b) := (ad = adr) \wedge mw \wedge mwb[b] \wedge \neg dbusy$$

in order to capture a write to byte b *of address* ad *and define the memory*

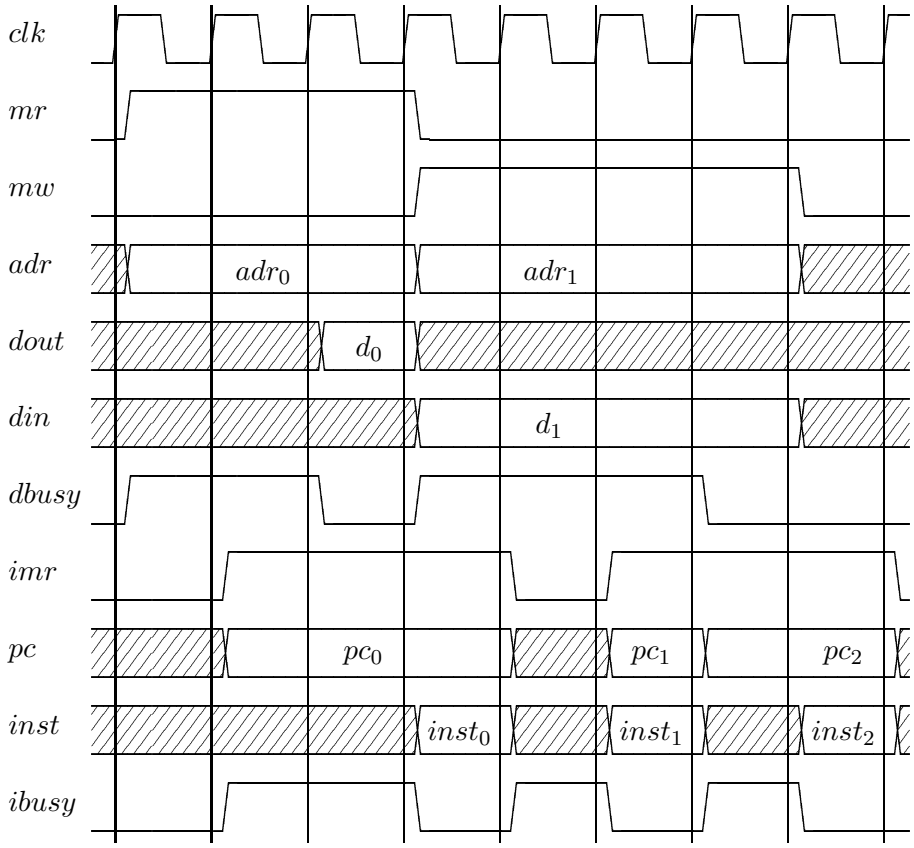

Figure 1.2: Timing of the memory interface

content M_I in cycle $t \in \mathbb{N}^+$ recursively as follows:

$$\begin{aligned} M_I^1 &:= init\_mem \\ \left|M_I^{t+1}[\langle ad\rangle]\right|_b &:= \begin{cases} \left|din^t\right|_b & \text{if } M_I.bw(ad,b)^t \\ \left|M_I^t[\langle ad\rangle]\right|_b & \text{otherwise} \end{cases} \end{aligned}$$

We call a memory interface ***correct*** *iff on valid input from the CPU according to definition 1.5.1, the following conditions hold $\forall t \in \mathbb{N}^+$:*

1. $mr^t \wedge \neg dbusy^t \implies dout^t = M_I[\langle adr^t\rangle]^t$ *(data cache consistency)*
2. $imr^t \wedge \neg ibusy^t \implies inst^t = M_I[\langle pc^t\rangle]^t$ *(instruction cache consistency)*
3. $\exists^{next}_{\neg dbusy}(t)$ *(data cache liveness)*
4. $\exists^{next}_{\neg ibusy}(t)$ *(instruction cache liveness)*

Thus, on concurrent read- and write accesses to the same address in the memory interface, there are two possible outcomes. Either the instruction read access terminates strictly after the write access and returns the correct data *after* the execution of the data write access, or it terminates in the same cycle or before the write and returns the *old* memory content. Both scenarios are equally possible in a correct memory interface.

Additionally, we note that according to lemma 1.2.10, the following equation trivially holds:

$$\left|M_I^t[\langle ad\rangle]\right|_b = \begin{cases} \left|din^{last_{M_I.bw(ad,b)}(t)}\right|_b & \text{if } \exists^{last}_{M_I.bw(ad,b)}(t) \wedge \\ & \quad last_{M_I.bw(ad,b)}(t) > 0 \\ |init\_mem[\langle ad\rangle]|_b & \text{otherwise} \end{cases} \tag{1.1}$$

A cache memory interface

The CPU clock in modern microprocessors runs at about ten times the frequency of the memory clock. Furthermore, typical main boards support only dynamic memory, i.e., SD-RAM or DDR-RAM. This results in an access latency of several slow memory cycles which in turn is equivalent to a latency of several dozens of fast CPU cycles. On the other hand, an ideal CPU accesses the memory at least once per cycle for the fetch of a new instruction. For memory instructions, we have two accesses. In case of super-scalar architectures, multiple instructions are actually fetched per cycle. Clearly, a naive implementation of a memory interface yields unbearably bad performance. Figure 1.3 illustrates this scenario.

Therefore, so-called *caches* are introduced in a memory interface. A small but fast memory called cache is added between CPU and main memory. Ideally, this cache uses the full CPU clock thus allowing for one cache access per CPU cycle as depicted in figure 1.4.

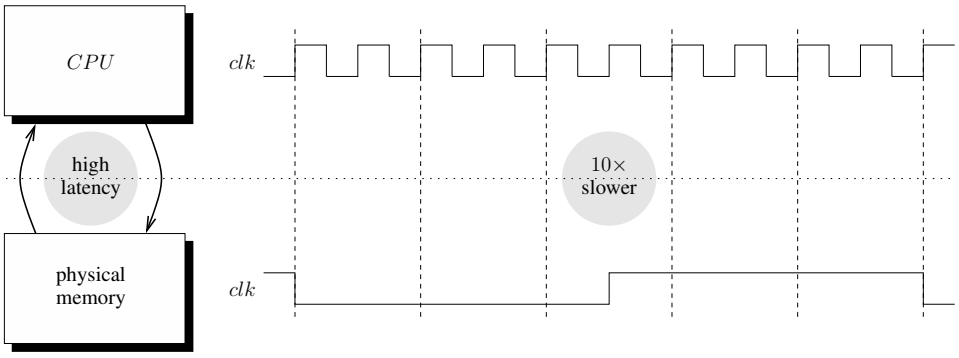

Figure 1.3: Primitive memory interface performance

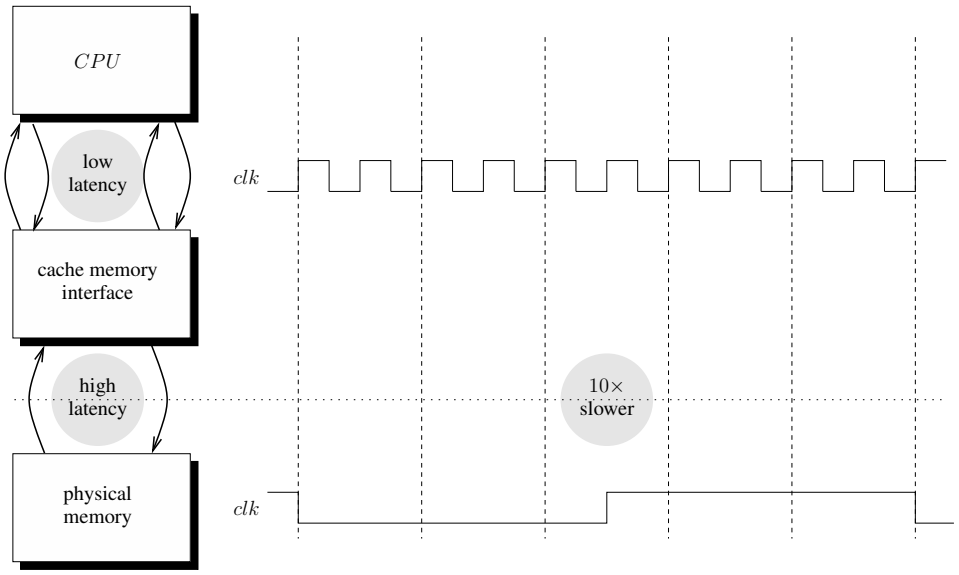

Figure 1.4: Split cache memory interface performance

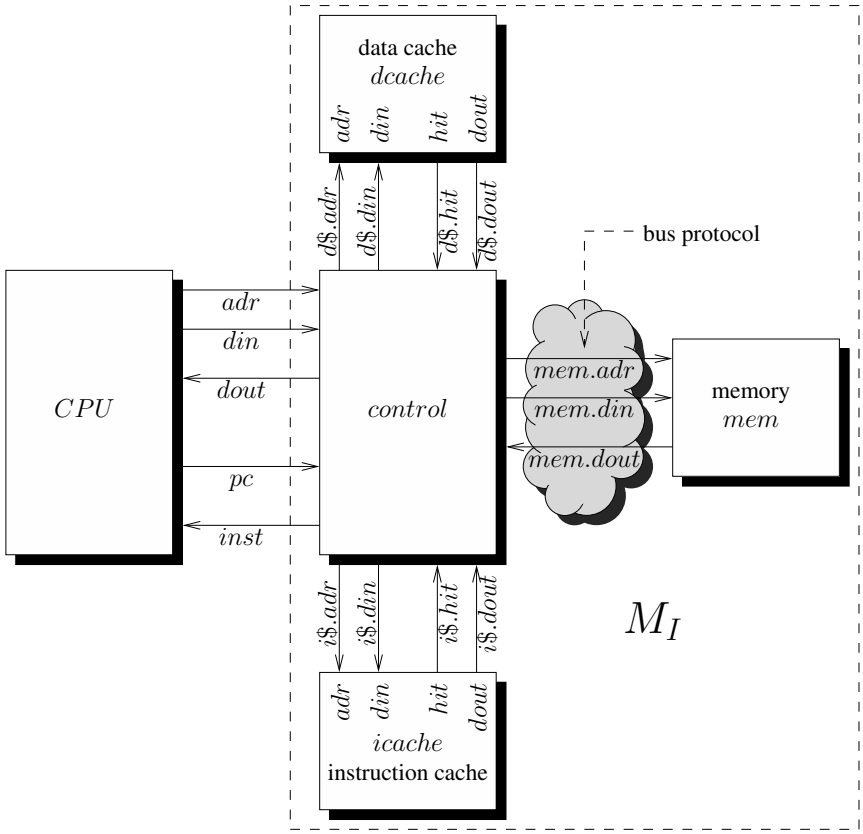

Figure 1.5: Data paths of a cache memory interface

If the memory access can be handled by the cache, for example a read access to a memory location currently held in the cache, the slow main memory access is avoided in favour of a fast cache access. This type of access is called a *hit*. If, on the other hand, the cache does not hold the desired data, it is loaded from the main memory and stored in the cache for possible further use. We call this a *miss*. Clearly, a miss performs no better than a memory access without any cache. Since a cache memory interface shows a performance superior to that of a simple memory interface only in case of a hit, the so-called *hit rate* of the cache becomes an important factor, i.e., the number of hits divided by the total number of cache accesses. This hit rate depends on the cache size, the code executed on the CPU, and several other factors. Common cache memory interface architectures have been shown to achieve hit rates of 95% and more on typical benchmark applications; hence, the additional cost for adding hardware caches usually pays off.

Signal	Description
Input	
$adr[a-1:0]$	memory word address
$din[8 \cdot B-1:0]$	data word input for cache data memory
$cdwb[B-1:0]$	byte write signals for data word input
vw	write enable for *valid* part of directory
val_in	data input for *valid* part of directory
tw	write enable for *tag* part of directory
dty	data input for *dirty* part of directory
dw	write enable for *dirty* part of directory
$\$rd$	initiates any cache access
$clear$	invalidates all the data in the cache
Output	
hit	signals valid data on $dout$ for input address adr
$dout[8 \cdot B-1:0]$	cache data output on address adr in case $hit = 1$
$dirty$	signals dirty data for input address adr
$ev[a-1:0]$	signals eviction address in case $dirty = 1$

Table 1.2: Description of input– and output signals of a cache

1.5.2 The cache consistency layer

In a pipelined microprocessor, there are up to two memory accesses per CPU cycle. Therefore, two separate caches are used in order to implement a cache memory interface, an instruction cache and a data cache. This allows for two simultaneous memory accesses per CPU cycle. Thus, in case the caches produce a hit, the CPU can really execute one instruction per CPU cycle. In order to keep the caches consistent and arbitrate between the memory accesses of the two caches, we introduce a control circuit. This circuit also decides whether a given request accesses the instruction cache, the data cache or the main memory. Figure 1.5 depicts this situation.

A cache memory interface has to fulfill the correctness criterion supplied in definition 1.5.2. Thus, a cache memory interface is transparent in the sense that the CPU does not really have to know anything about the implementation of the memory interface, be it simple or with split caches.

Similar to definition 1.5.2 of correct memory interface, we introduce the notion of consistent caches in this section. As outlined in the previous sections, caches signal valid data with an active hit signal. Hence, a straightforward cache consistency property states that in cases of a hit, the cache returns the same data a consistent memory would—namely, the data that was last written to the read address. Table 1.2 summarizes all cache input and output signals in order to allow for a formal definition of cache consistency. Note that we currently only refer to adr, din, $dout$, and $cdwb$; all the

other signals are introduced in chapter 2.

Definition 1.5.3 *For any $ad \in \mathbb{B}^a$ and $b < B$, we introduce a parameterized predicate on the cache input that captures writes to byte b of address ad by defining $\$.bw(ad, b) := (adr = ad) \wedge cdwb[b]$.*

A cache is called ***consistent*** *iff the following properties hold for any cycle $t \in \mathbb{N}^+$ and any $b \in \mathbb{Z}_B$:*

$$hit^t \implies \exists^{last}_{\$.bw(adr^t,b)}(t) \wedge last_{\$.bw(adr^t,b)}(t) > 0 \wedge \left|dout^t\right|_b = \left|din^{last_{\$.bw(adr^t,b)}(t)}\right|_b$$

In the following chapter, we will show different cache implementations to fulfill the above cache consistency property and some other additional properties. In chapter 3, we then use abstract instruction- and data caches obeying the above definition in order to implement a cache memory interface and prove it correct according to definition 1.5.2. Finally, chapter 4 only uses the definition of a correct memory interface in order to prove a whole CPU correct. Thanks to clean interfaces, putting these three proofs together to a single proof of the overall correctness of a pipelined out-of order CPU with a cache memory interface comprising split instruction- and data caches does not require any additional proof effort.

Chapter 2

Caches

In this chapter we will establish implementations and correctness proofs for different implementations of caches. Since we exclusively deal with caches in this chapter, we will use the shorthand notation $bw(ad, b)$ for $\$.bw(ad, b)$ that was introduced in definition 1.5.3 in the previous chapter.

2.1 Definition

A cache is a small memory containing a subset of the data of the main memory. The basic idea is that an access to this small cache memory is fast as opposed to the slow off-chip main memory. The cache signals that it holds valid data for an address *adr* by raising a *hit* signal. If, on the other hand, the cache produces a miss, i.e., it does not raise the *hit* signal, an access to the main memory is required. The required data is read from the main memory and stored in the cache. This is done due to the heuristic *principle of temporal locality*: If a CPU accesses an address *adr*, it is likely that it accesses the same memory location *adr* once again later on. By copying this data to the cache, a future access to the same address thus becomes fast.

If the CPU accesses an address *adr*, it is also likely that it accesses an address close to *adr* later on, i.e., $adr + 1$. This is called the *principle of spatial locality*. Therefore, on a miss, not only one word is read from the main memory and stored in the cache, but a whole so-called *cache line*. A cache line consists of 2^s data words, where $s \in \mathbb{N}$ is a cache parameter. Note that data words are called *cache sectors* in [MP00]. The transfer of a cache line from the main memory to the cache is called a *line fill*. For $s = 0$, a cache line just equals one word. For $s > 0$, the cache line consists of multiple data words and we call the cache *sectored*. One further reason for the use of sectored caches is the fact that usually, the main memory consists of dynamic RAM which allows for fast accesses to consecutive addresses called bursts.

In case of a cache miss, it is possible that the cache is full, i.e., all the cache memory contains only valid data and the cache cannot hold the additional

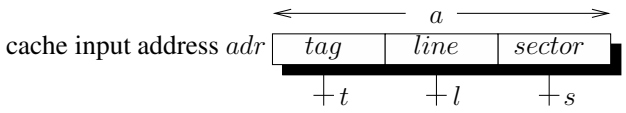

Figure 2.1: Partitioning of an a-bit cache address

cache line that is to be added in a line fill. In this case, some valid cache line has to be evicted from the cache in order to make room for the new cache line. We perform this eviction in *two* steps, i.e., we first *invalidate* the 'old' cache line and only then perform a line fill for the *new* cache line. The process of selecting an appropriate cache line for invalidation is called *replacement policy*. Different replacement policies are described later on.

An important aspect in designing caches is the mapping of memory locations to possible cache locations. In general, a memory location is mapped to a set of cache locations that can hold the data of the corresponding memory location. In case of software caches, any memory location usually can be mapped to any cache location. This is called *full associativity*. In a hardware implementation of a cache, however, full associativity is too expensive except for very small caches, e.g., caches with less than 64 entries that are used for translation look-aside buffers (TLBs).

In common hardware caches, memory locations are mapped to sets of cache locations that all have the same cardinality. We therefore introduce an additional cache parameter K equaling this cardinality. We distinguish three types of caches according to the value of K. In the most simple case, $K = 1$, we call the cache *direct mapped*. For $K \geq 2$, the cache is called *K-way set associative*; it basically consists of K so-called *ways* which are direct mapped caches themselves. If K equals the number of cache lines the cache memory can hold, the cache is actually fully associative.

In order to specify the size of the cache data memory, we introduce a cache parameter l such that a single cache way can hold 2^l different cache lines. This equals a total of $K \cdot 2^{l+s}$ words since a cache line consists of 2^s words and the cache consists of K ways. Additionally, we set $t := a - l - s$. A cache input address adr is split into the three parts *tag*, *line*, and *sector* according to figure 2.1. Two memory addresses that only differ in their tag part are mapped to the same set of K cache locations. Any two addresses differing only in the sector part belong to the same cache line, thus offering the specified line size of 2^s data words.

Definition 2.1.1 *Let $n \in \mathbb{N}^+$ and $i, j \in \mathbb{Z}_n$ with $i > j$. We then introduce for any $a, b \in \mathbb{B}^n$ the following shorthand notations:*

$$\begin{aligned} a =_j^i b &:\Longleftrightarrow a[i-1:j] = b[i-1:j] \\ a =_j b &:\Longleftrightarrow a =_j^n b \\ [a]_j &:= \{a' \in \mathbb{B}^n | a =_j a'\} \end{aligned}$$

Parameter	Meaning
B	number of bytes in a data word
a	address width of the memory, $a = t + l + s$
t	tag width
l	cache way holds 2^l lines
s	2^s words per cache line
K	associativity

Table 2.1: Cache parameters

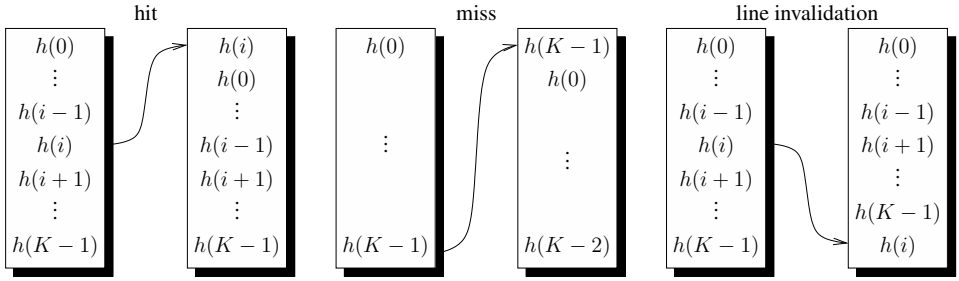

Figure 2.2: LRU history updates

Thus, we denote that two addresses adr and adr' belong to the same cache line by $adr =_s adr'$. They are mapped to the same set of cache locations iff $adr =_0^{l+s} adr'$ and they access the same cache directory entry iff $adr =_s^{l+s} adr'$. The different cache parameters are summarized in table 2.1. Note that one parameter out of a, t, l, and s is actually redundant.

For each of its 2^l cache lines, a cache way needs additional memory in order to store if the cache line holds valid data and to store the actual memory address of this data. Therefore, a each cache way contains a *directory* entry for each of its cache lines comprising a valid flag and the tag of the corresponding memory address in case the data is valid. Since the line and sector part of a memory address equal the corresponding parts in a cache address that it is mapped to, it suffices to store only the tag part in the directory in order to identify the complete memory address.

For caches with $K \geq 2$, the replacement policy becomes an important topic. There are different replacement policies for caches, e.g., random replacement where the line to be evicted is selected randomly. In this thesis, we consider *least recently used* (LRU) policy. As the name suggests, the cache line that was used least recently is replaced. The rationale behind is that heuristically, the line that was used least recently is not accessed again in the near future and thus, we do not have to pay the extra penalty of fetching the line again we just wrote back to the main memory.

In order to implement least recently used policy, we need additional data that is called a *history vector*. A history vector is a permutation $h : \mathbb{Z}_K \rightarrow$

$\mathbb{Z}_K$ that maps positions in the cache access history to cache ways. Thus, $h(0)$ is the index of most recently used way, and $h(K-1)$ the index of the least recently used way. Since a cache holds $2^l \cdot K$ lines, we have 2^l different history vectors in a K-way set-associative cache. Depending on the type of a cache access, this history vector is updated according to figure 2.2. On a hit, the hit way i becomes the most recently used way. On a miss, the least recently used way is evicted and new data is placed in this way; therefore, it becomes the most recently used way. If a cache line is to be invalidated, the hit way i becomes least recently used since it no longer contains valid data; thus, it can be safely overwritten in the next miss access.

In case of a write access, a cache memory interface implementation with underlying caches as characterized above would have to update the cache memory *and* the slow main memory. In order to increase performance [MK00], caches therefore often employ so-called *write back* policy, i.e., in case of a write access, the data is written to the cache, it is marked as *dirty*, but *not* written back to the main memory. In case such a dirty line is to be evicted from the cache, one has to pay the penalty for this policy by writing back the dirty line to the main memory before replacing it by the new data. In order to keep track of dirty lines, the cache directory is thus extended by a *dirty* bit for each cache line. Additionally, we extend the cache by an address output *ev* equalling the memory address that dirty data is to be written back to.

Note that we can have dirty data both on a hit and on a miss. In case of a hit, an active *dirty* just signals that the hit cache line contains dirty data, i.e., it needs to be written back in case of a line invalidation access. In this case, the eviction address *ev* we added as output of the cache just returns the input address itself. In case of a miss, on the other hand, *dirty* signals that the cache line that is to be evicted contains dirty data. In this case, *ev* returns the address that the dirty data of the way that is to be evicted has to be written back to. Hence, the additional output *ev* is actually needed only in case of a miss, but not on a hit.

2.2 Correctness criteria

Let us first specify the interface of a cache. We consider parametrized caches; i.e., the address width is a bits and a data word in the cache RAM contains B bytes. All the input- and output signals of a cache are summarized in table 1.2 on page 22. As inputs, we obviously have an address input *adr*, data input *din* as well as separate byte write signals $cdwb[B-1:0]$ for the B bytes in *din*. Additional input signals are write signals for the the different parts of the cache directory, i.e., *vw*, *tw*, and *dw* for valid, tag, and dirty, respectively. The signals *val_in* and *dty* are the inputs of the valid and dirty part of the directory, respectively. An additional signal $\$rd$ signals the

beginning of a cache access, i.e., a read or write access. Last but not least, there is a *clear* input that initializes the cache directory.

As outputs, we have the *hit* signal and the output data *dout* as well as *dirty* and *ev* as introduced above. Based on these sets of signals, we will now specify the correctness of a cache.

The basic property we demand of any cache is consistency on a hit, i.e., the data output in case of a hit equals the last data written to the hit address. This property is formally captured in definition 1.5.3 on page 23.

Note that cache consistency as defined previously does not cover the whole cache content, i.e., consistency in cycle t is only claimed for input address adr^t since both hit^t and $dout^t$ depend on the current address in cycle t. We therefore use definition 1.2.7 on page 6 in order to introduce a consistency criterion based on hit^t_{ad} and $dout^t_{ad}$.

Definition 2.2.1 *A cache is called* ***extended consistent*** *iff the following properties hold for any cycle $t \in \mathbb{N}^+$, any address $ad \in \mathbb{B}^a$, and any byte $b \in \mathbb{Z}_B$:*

$$hit^t_{ad} \implies \exists^{last}_{bw(ad,b)}(t) \wedge last_{bw(ad,b)}(t) > 0 \wedge \left|dout^t_{ad}\right|_b = \left|din^{last_{bw(ad,b)}(t)}\right|_b$$

Note that $last_{bw(ad,b)}(t)$ may have different values for every $b < B$. Furthermore, a hit signals valid data for all bytes in the whole cache line. Thus, cache consistency claims that the whole cache line has been written prior to the hit. Therefore, cache consistency does not hold for arbitrary input sequences. For example, we need that on a line fill, a whole cache line is written to the cache.

According to table 1.2, a cache access is initiated by an active $\$rd$. When arguing about cache accesses, we need a formal way to state that two cycles belong to the same cache access. We therefore introduce a predicate *crd* on the cache input by $crd := \$rd \vee clear$. We say that two cycles t and t' belong to the same cache access if $last_{crd}(t) = last_{crd}(t')$. The following definition summarizes all the assumptions we have to make on the input of a cache in order to prove it consistent.

Definition 2.2.2 *A cache input is called* ***valid*** *if it fulfills the predicate valid_input? given by the following conditions:*

1. $clear^0$
2. $\forall t \in \mathbb{N}^+ : vw^t \wedge val\_in^t \implies tw^t$
3. $\forall t \in \mathbb{N}^+ : vw^t \vee tw^t \vee dw^t \vee \exists l \in \mathbb{Z}_B : cdwb^t[l] \implies adr^t =_s adr^{last_{crd}(t)} \wedge \neg clear^{last_{crd}(t)} \wedge \neg\$rd^t$
4. $\forall t \in \mathbb{N}^+ : tw^t \implies vw^t \wedge val\_in^t$

5. $$\begin{aligned}&\forall t \in \mathbb{N}^+ : vw^t \wedge val\_in^t \implies \\ &\quad \exists j \in]last_{crd}(t) : t[: vw^j \wedge \neg val\_in^j \wedge \\ &\quad (\forall k \in]last_{crd}(t) : j[: \neg vw^k) \wedge (\forall k \in]j : t[: \neg vw^k) \wedge \\ &\quad (\forall ad \in \left[adr^t\right]_s, b \in \mathbb{Z}_B : \exists t' \in]j : t] : bw(ad,b)^{t'})\end{aligned}$$

6. $$\begin{aligned}&\forall t \in \mathbb{N}^+ : \exists l \in \mathbb{Z}_B : cdwb^t[l] \wedge \neg hit^{last_{crd}(t)} \implies \\ &\quad \exists j \in]last_{crd}(t) : t[: vw^j \wedge \neg val\_in^j\end{aligned}$$

Note that $\exists^{last}_{crd}(t)$ holds for all $t \in \mathbb{N}^+$ on a valid cache input because of condition 1. Apart from the initial *clear* and some restrictions on the write signals for the cache directory, we demand in condition 3 that during a cache access initiated by $\$rd^t$, there is no write access to an address outside the cache line given by the address in the cycle of the beginning of the access, i.e., $adr^{last_{crd}(t)}$. In particular, this item also ensures that there is no active write signal in an access 'initiated' by a clear signal, i.e., $clear^{last_{crd}(t)}$, and at the beginning of an access, i.e., during a $\$rd$. Note that we allow read accesses to different cache lines during one cache access. We need these accesses in order to implement write-back caches.

Furthermore, condition 5 states that any line fill is complete, i.e., given the end of a line fill in cycle t, i.e., $vw^t \wedge val\_in^t$, the line is invalidated in a cycle j of the same access and between these cycles j and t, all bytes in a cache line are written as indicated by predicate $bw(ad,b)$ for any byte b and any address ad in the same cache line. We also demand in condition 6 that any byte write after a cache miss is preceded by an invalidation of the same cache line for otherwise, cache consistency would be violated since valid cache data may be overwritten by such a write.

Apart from this central cache consistency, we need a few additional properties in order to verify that our memory interface with split caches is consistent. These properties mainly cover comparatively simple facts, i.e., if we do not have an hit on address ad in cycle t, but on a later cycle t', a line fill has occurred in between. Accordingly, if a hit is lost somewhere between cycle t and t', the corresponding line was invalidated in between. Because of the comparative simplicity of the following properties, we will not give full details on their verifications in the following sections.

Definition 2.2.3 *We call a cache* ***control-consistent*** *if it is consistent and the following properties hold:*

1. *The hit signal only changes due to a vw or clear input, i.e.,*

$$\begin{aligned}&\forall t \in \mathbb{N}^+, t' \in [last_{crd}(t) : t[: hit^t \neq hit^{t'} \wedge adr^t =_s adr^{t'} \implies \\ &\quad \exists k \in [t' : t[: vw^k \wedge val\_in^k = hit^t \vee clear^k \wedge hit^{t'}.\end{aligned}$$

2. *An input signal tw creates a hit, i.e.,*

$$\forall t \in \mathbb{N}^+ : tw^t \wedge \neg clear^t \wedge adr^t =_s adr^{t+1} \implies hit^{t+1}.$$

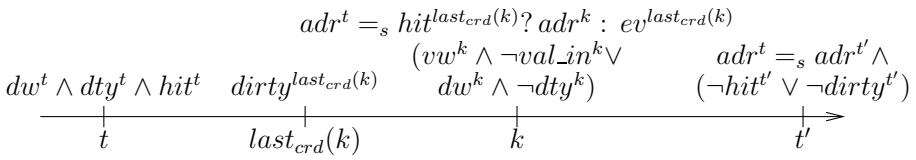

Figure 2.3: Illustration of *dirty* consistency

3. *If a cache signals dirty data at the beginning of a cache access and is accessed on the eviction address later on in the same access without any writes to the dirty or tag RAM in between, it signals a dirty hit, i.e.,*

$$adr^t =_s ev^{last_{crd}(t)} \wedge \neg clear^{last_{crd}(t)} \wedge dirty^{last_{crd}(t)} \wedge (\forall t' \in]last_{crd}(t) : t[: \neg dw^{t'} \wedge \neg vw^{t'}) \implies hit^t \wedge dirty^t.$$

4. *The dirty flag in the cache directory is consistent. More formally, if a hit cache line on address adr^t is marked as dirty in cycle t, but is either no longer hit or dirty in a later cycle t', we find an intermediate cycle k where either clear was active, the cache line was invalidated, or marked as clean. This is illustrated in figure 2.3. In particular, if the cache line was invalidated or marked as clean in cycle k, we can also conclude that k was either in the same cache access as t itself or the cache line was still dirty at the beginning of the cache access, i.e., $dirty^{last_{crd}(k)}$, which is also illustrated in figure 2.3. In addition, we can conclude that the address adr^t is in the same cache line as adr^k in case of a hit, while it is in the same cache line as the eviction address at the beginning of the access otherwise. Formally, $\forall t \in \mathbb{N}^+, t' \in \mathbb{Z}_{\geq t}$:*

$$\begin{aligned} adr^t =_s adr^{t'} \wedge dw^t \wedge dty^t \wedge hit^t \wedge \neg vw^t \wedge \neg clear^t \wedge \\ (\neg dirty^{t'} \vee \neg hit^{t'}) \implies \exists k \in]t : t'[: clear^k \vee \\ \left(dw^k \wedge \neg dty^k \vee vw^k \wedge \neg val\_in^k\right) \wedge \\ \left(last_{crd}(k) \leq t \vee dirty^{last_{crd}(k)} \wedge \right. \\ \left. adr^t =_s \left(hit^{last_{crd}(k)}? adr^k : ev^{last_{crd}(k)}\right)\right). \end{aligned}$$

5. *On a cache hit, the eviction address ev equals the input address adr as we outlined in our introduction of write back caches.*

$$hit^t \implies ev^t = adr^t$$

6. *The cache stays hit continually from the cycle of the last write as illustrated in figure 2.4, i.e., on a hit in cycle t on address adr^t, we find a cycle m in the same cache access as the last write to the hit cache line such that the cache signals a hit when addressed in the same cache line*

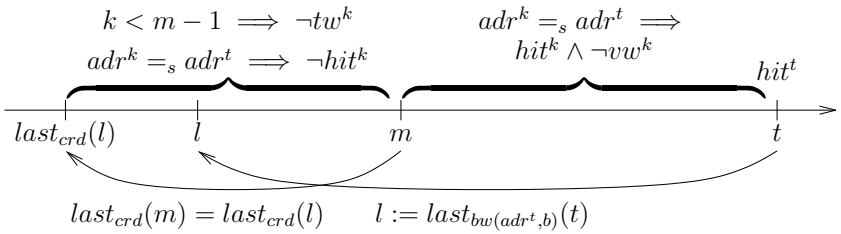

Figure 2.4: Illustration of the continuous hit property

as adr^t from cycle m on. In addition, we can conclude that the valid part of the directory for the cache line containing adr^t was not written between m and t.

$$\begin{gathered} hit^t \implies \exists^{last}_{bw(adr^t,b)}(t) \wedge last_{bw(adr^t,b)}(t) > 0 \wedge \\ \exists m \in [last_{bw(adr^t,b)}(t) : t] : last_{crd}(m) = last_{crd}(last_{bw(adr^t,b)}(t)) \wedge \\ \forall k \in [last_{bw(adr^t,b)}(t) : t] : \big(adr^k =_s adr^t \implies \big(hit^k \iff k \geq m\big)\big) \wedge \\ \big(k \in [m : t[\implies \neg\big(vw^k \wedge adr^k =_s adr^t\big)\big) \wedge \big(k < m-1 \implies \neg tw^k\big)\,. \end{gathered}$$

7. *A hit can only be created by an input signal tw to a corresponding address, i.e.,*

$$\begin{gathered} \forall t > 0, t' > t : adr^t =_s adr^{t'} \wedge \big(\neg hit^t \vee vw^t \wedge \neg val\_in^t\big) \wedge hit^{t'} \implies \\ \exists k \in [t : t'[: tw^k \wedge adr^k =_s adr^t. \end{gathered}$$

2.3 A direct-mapped cache

The easiest and cheapest hardware implementation of a cache is a direct mapped cache as depicted in figure 2.5. Such a direct mapped cache simply contains an array *data* of B banks of $2^{l+s} \times 8$ data RAM that is addressed by the line and sector part of the address. A hit occurs if the tag of the address matches the tag in the directory and the directory entry is valid. Dirty data is signalled only if the corresponding directory entry is both valid and dirty. As eviction address ev, the direct mapped cache simply uses the old address where the tag part is replaced by the corresponding directory's tag.

Formally, we can specify the outputs of a direct mapped cache as follows. Note that we abbreviate $adr[l+s-1:s]$ by adr_l. Bit i of the data word *dout* is located in bank i div 8 and equals bit i mod 8 from this bank. Hence, we use a λ-expression in order to compute *dout* and first select the array entry i div 8 of array *data*, then use address $adr[l+s-1:0]$ in order to read a byte stored in this RAM, and finally select bit i mod 8 in this byte.

$$\begin{array}{rcl} hit & = & valid[adr\_l] \wedge eq_t(tag[adr\_l], adr[a-1:l+s]) \\ dout & = & \lambda_{i \in \mathbb{Z}_{8 \cdot B}} data[i \text{ div } 8]\,[adr[l+s-1:0]]\,[i \bmod 8] \\ dirty & = & valid[adr\_l] \wedge dirty[adr\_l] \\ ev & = & tag[adr\_l] \cdot adr\_ls \end{array} \tag{2.1}$$

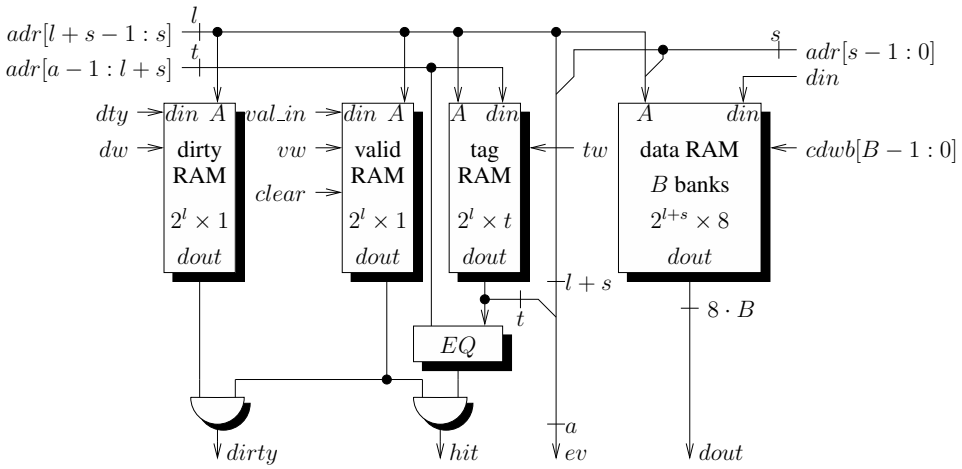

Figure 2.5: Direct mapped cache

Furthermore, note that

$$|dout|_b = data[b][adr[l+s-1:0]]$$

trivially holds. Using primed notation, the next state of the direct mapped cache according to figure 2.5 is given by:

$$\begin{aligned}
valid' &= \lambda_{ad\in\mathbb{B}^l} \begin{cases} 0 & clear \\ val_in & \neg clear \wedge vw \wedge (ad =_s^{l+s} adr) \\ valid[ad] & \text{otherwise} \end{cases} \\
dirty' &= \lambda_{ad\in\mathbb{B}^l} \begin{cases} dty & dw \wedge (ad =_s^{l+s} adr) \\ dirty[ad] & \text{otherwise} \end{cases} \\
tag' &= \lambda_{ad\in\mathbb{B}^l} \begin{cases} adr[a-1:l+s] & tw \wedge (ad =_s^{l+s} adr) \\ tag[ad] & \text{otherwise} \end{cases} \\
data' &= \lambda_{b\in\mathbb{Z}_B}\lambda_{ad\in\mathbb{B}^{l+s}} \begin{cases} |din|_b & cdwb[b] \wedge (ad =_0^{l+s} adr) \\ data[b][ad] & \text{otherwise} \end{cases}
\end{aligned} \tag{2.2}$$

2.3.1 Correctness proof

Although a direct mapped cache only contains two *and*-gates and an *equal*-tester in addition to its RAM, the formal verification of its consistency as previously defined is not trivial.

First of all, we note that cache consistency by definition 1.5.3 is not an inductive invariant. Consider the case where the address changes, i.e., $adr^t \neq adr^{t+1}$. In order to show cache consistency for cycle $t+1$, one would like to use cache consistency for cycle t. However, cache consistency in cycle t

only covers address adr^t, while we need cache consistency for address adr^{t+1} in cycle t as a precondition. Therefore, we introduced the *extended* cache consistency by definition 2.2.1 which covers the above mentioned case in the induction step since consistency is claimed for all possible input addresses.

As a first step towards verifying extended cache consistency for the direct mapped cache, we show the first part of the claim in the extended consistency predicate, i.e., on a hit, all the bytes in the cache line have previously been written to the cache.

Lemma 2.3.1 *If a direct mapped cache does not signal a hit on address ad in cycle t, but in cycle $t+1$, the signal tw was active in cycle t and the input address in cycle t and ad belong to the same cache line. Formally, $\forall t \in \mathbb{N}^+$ and $ad \in \mathbb{B}^a$:*

$$valid\_input? \wedge hit_{ad}^{t+1} \wedge \neg hit_{ad}^{t} \implies tw^t \wedge adr^t =_s ad$$

Proof: We introduce the shorthand notation ad_l for $ad[l+s-1:s]$ and conclude from hit_{ad}^{t+1} and $\neg hit_{ad}^{t}$, from the cache output implementation given by equation (2.1) and definition 1.4.1 of the equal-tester

$$\begin{aligned} &valid^{t+1}[ad\_l] \wedge (tag^{t+1}[ad\_l] = ad[a-1:l+s]) \\ &\quad \neg valid^{t}[ad\_l] \vee (tag^{t}[ad\_l] \neq ad[a-1:l+s]) \end{aligned}$$

From items 2 and 4 of $valid\_input?$, we conclude that $tw^t \iff vw^t \wedge val\_in^t$. We now split cases on tw^t.

1. In case $\neg tw^t$ holds, we conclude $tag^{t+1}[ad\_l] = tag^t[ad\_l]$ by equation (2.2). Thus, $tag^t[ad\_l] = ad[a-1:l+s] \wedge \neg valid^t[ad\_l]$ holds. Since $valid^{t+1}[ad\_l]$ holds, we can conclude $vw^t \wedge val\_in^t$ which is a contradiction because of $valid\_input?$.

2. If, on the other hand, tw^t holds, then $vw^t \wedge val\_in^t$ also holds. Additionally, we know $tag^{t+1}[ad\_l] = adr^t[a-1:l+s]$ and $ad =_s^{l+s} adr^t$ by equation (2.2). Because of hit_{ad}^{t+1}, we then have $ad =_{l+s} adr^t$ which leads to $ad =_s adr^t$ and thus concludes the claim. □

Lemma 2.3.2 *If a direct mapped cache signals a hit in cycle t, but not in a previous cycle t' for the same address, there exists a cycle t'' between t' and t where vw and val_in are active for an address in the same cache line. Formally, we have $\forall t \in \mathbb{N}^+$, $t' \in \mathbb{Z}_t$ and $ad \in \mathbb{B}^a$:*

$$valid\_input? \wedge hit_{ad}^{t} \wedge \neg hit_{ad}^{t'} \implies \exists t'' \in [t' : t[: vw^{t''} \wedge val\_in^{t''} \wedge adr^{t''} =_s ad$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Because of $valid\_input?$, $clear^0$ holds and thus, $valid^1 = \lambda_{ad \in \mathbb{B}^l} 0$ by construction of the direct mapped cache. Therefore, hit_{ad}^1 cannot hold which concludes the induction base.

Induction step ($\mathbf{t \to t+1}$): Let $valid\_input?$, hit_{ad}^{t+1} and $\neg hit_{ad}^{t'}$ hold. We have to show $\exists t'' \in [t' : t+1[: vw^{t''} \wedge val\_in^{t''} \wedge adr^{t''} =_s ad$. We split cases on hit_{ad}^{t}.

1. Let hit_{ad}^{t} hold. By induction hypothesis, we conclude $\exists t'' \in [t' : t[: vw^{t''} \wedge val\_in^{t''} \wedge adr^{t''} =_s ad$. This concludes the claim since $[t' : t[\subseteq [t' : t+1[$ holds.

2. Let $\neg hit_{ad}^{t}$ hold. By applying lemma 2.3.1, we conclude $tw^t \wedge adr^t =_s ad$. Because of $valid\_input?$, we can conclude $vw^t \wedge val\_in^t$ from tw^t which concludes the claim. □

With this helper lemma and a valid cache input, we are able to conclude that all bytes in the cache line have been written on a hit.

Lemma 2.3.3 *In case of a hit in cycle t, all the bytes in the cache line have been written before cycle t, i.e., $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$:*

$$valid\_input? \wedge hit_{ad}^{t} \implies \exists_{bw(ad,b)}^{last}(t) \wedge last_{bw(ad,b)}(t) > 0$$

Proof: Because of predicate $valid\_input?$, we have $clear^0$ which implies $\neg hit_{ad}^{1}$. Thus, we can apply lemma 2.3.2 to cycles 1 and t in order to conclude $\exists t' \in [1 : t[: vw^{t'} \wedge val\_in^{t'} \wedge adr^{t'} =_s ad$. Predicate $valid\_input?$ implies that the line fill that ended in cycle t' is complete, i.e., $\exists j \in]last_{crd}(t') : t'[: vw^j \wedge \neg val\_in^j$ and we have

$$\forall ad' \in \left[adr^{t'}\right]_s, b' \in \mathbb{Z}_B : \exists l \in]j : t'] : bw(ad', b')^l.$$

Since $ad =_s adr^{t'}$, we can instantiate this formula with ad and b in order to get a cycle $l > 0$ with $bw(ad, b)^l$. With proposition 1.2.9, we conclude $\exists_{bw(ad,b)}^{last}(t) \wedge last_{bw(ad,b)}(t) \geq l$ which concludes the claim. □

Thus we have show the first part of extended cache consistency for a direct mapped cache. We now proceed with another easy lemma about the content of the data RAM.

Lemma 2.3.4 *The content of the cache data RAM only change on a byte write, i.e., $\forall ad \in \mathbb{B}^a, b \in \mathbb{Z}_B, t \in \mathbb{N}, t' \in \mathbb{Z}_{\geq t}$:*

$$\left|dout_{ad}^{t'}\right|_b \neq \left|dout_{ad}^{t}\right|_b \implies \exists t'' \in [t : t'[: cdwb^{t''}[b] \wedge ad =_0^{l+s} adr^{t''}$$

We omit the proof of this lemma because it is just a simple induction.

Lemma 2.3.5 *If a cache does not signal a hit on address ad in cycle t, extended cache consistency holds for this particular address ad in cycle $t+1$, i.e., $\forall ad \in \mathbb{B}^a, b \in \mathbb{Z}_B, t \in \mathbb{N}^+$:*

$$\begin{array}{l} valid\_input? \wedge \neg hit_{ad}^{t} \wedge hit_{ad}^{t+1} \implies \\ \quad \exists_{bw(ad,b)}^{last}(t+1) \wedge last_{bw(ad,b)}(t+1) > 0 \wedge \\ \quad\quad \left|dout_{ad}^{t+1}\right|_b = \left|din^{last_{bw(ad,b)}(t+1)}\right|_b \end{array}$$

Proof: With lemma 2.3.3, we can easily conclude $\exists^{last}_{bw(ad,b)}(t+1)$ and $last_{bw(ad,b)}(t+1) > 0$. We set $l := last_{bw(ad,b)}(t+1)$. By applying lemma 2.3.1, we also conclude $tw^t \wedge adr^t =_s ad$. Furthermore, $vw^t \wedge val\_in^t$ holds because of property 4 of $valid\_input?$. We now split two cases.

1. The last write to byte b of address ad occurred *after* the last $\$rd$, i.e., $l > last_{crd}(t+1)$. Note that $bw(ad,b)^l$ and $\neg bw(ad,b)^{t'}$ for any $t' \in]l:t[$ both hold by proposition 1.2.9 as well as $last_{crd}(l) = last_{crd}(t)$. Therefore,
$$\left|dout_{ad}^{l+1}\right|_b = \left|din^l\right|_b$$
also holds by construction of the direct mapped cache. We show the remaining claim, i.e.,
$$\left|dout_{ad}^{t+1}\right|_b = \left|dout_{ad}^{l+1}\right|_b$$
by contradiction. Using lemma 2.3.4 for cycles $t+1$ and $l+1$, we thus find a write to the data RAM, i.e.,
$$cdwb^{t''}[b] \wedge ad =_0^{l+s} adr^{t''}$$
for a cycle $t'' \in [last_{bw(ad,b)}(t+1)+1 : t+1[$. Note that proposition 1.2.9 guarantees $last_{crd}(t'') = last_{crd}(t)$. Because of $\neg bw(ad,b)^{t''}$, we have
$$\neg\left(cdwb^{t''}[b] \wedge ad =_s adr^{t''}\right).$$
In other words, we have $ad \neq_s adr^{t''}$. Since both in cycle l and in t'', some write signal is active and $last_{crd}(t'') = last_{crd}(l)$ holds, item 3 of $valid\_input?$ guarantees $adr^t =_s adr^{t''}$ which is a contradiction. Hence, this case of the claim is proved.

2. The last write to byte b of address ad did not occur after the last $\$rd$, i.e., $l \leq last_{crd}(t+1)$. We show that this case cannot occur by contradiction. Item 5 of $valid\_input?$ guarantees the completeness of the line fill that is finished in cycle t by $vw^t \wedge val\_in^t$, i.e., $\exists j \in]last_{crd}(t) : t[: vw^j \wedge \neg val\_in^j \wedge \forall ad \in \left[adr^t\right]_s, b \in \mathbb{Z}_B :$
$\exists k \in]j:t] : bw(ad,b)^k$. Since $ad =_s adr^t$ also holds, we have $bw^k(ad,b)$ for some $l \in]last_{crd}(t)+1 : t]$. Therefore, $last_{bw(ad,b)} \geq k$ holds which is a contradiction and thus finishes the proof. □

In order to show extended consistency, we need a couple of additional helper lemmas which we will introduce below. In particular, we have to focus on the case that the cache signals a hit in cycle t and some data write also occurs in cycle t *not* to the hit address, but to some different address that is mapped to the same location in the data memory and thus would destroy data consistency.

Lemma 2.3.6 *If the direct mapped cache signals a hit, its tag part of the directory is not changed throughout the whole cache access, i.e.,* $\forall ad \in \mathbb{B}^a, t \in \mathbb{N}^+, t' \in \mathbb{Z}_{\geq t}$*:*

$$valid\_input?^t \wedge last_{crd}(t) = last_{crd}(t') \wedge hit_{ad}^t \wedge ad =_s adr^{last_{crd}(t)} \implies tag^t = tag^{t'}$$

Proof: Let $valid\_input?^t$ hold. We fix an arbitrary $ad \in \mathbb{B}^a$ and $t \in \mathbb{N}^+$ with $ad =_s adr^{last_{crd}(t)}$ and hit_{ad}^t; the remaining claim

$$last_{crd}(t) = last_{crd}(t') \implies tag^t = tag^{t'}$$

is shown by induction on t'.
Induction base ($\mathbf{t' = t}$): For $t' = t$, the right-hand side of the claim is trivially fulfilled.
Induction step ($\mathbf{t' \to t' + 1}$): Let $last_{crd}(t) = last_{crd}(t' + 1)$ hold. We then have to show $tag^t = tag^{t'+1}$. By properties 1.2.9 of *last*, we conclude that $last_{crd}(t) = last_{crd}(t')$ also holds. Thus, we can apply the induction hypothesis in order to get $tag^t = tag^{t'}$. We then show the remaining claim $tag^{t'} = tag^{t'+1}$ by contradiction, i.e., assuming inequality. Let therefore $tw^{t'}$ hold and $adr^t[a-1:l+s] \neq tag^{t'}[adr^{t'}[l+s-1:s]]$. By hit_{ad}^t, we conclude

$$tag^t[ad[l+s-1:s]] = ad[a-1:l+s] \wedge valid^t[ad[l+s-1:s]]$$

and with item 3 of $valid\_input?^t$, we additionally get $adr^{t'} =_s adr^{last_{crd}(t')}$. This leads to $adr^t =_s ad$ and with $tag^t = tag^{t'}$ also to $adr^t \neq_{l+s} ad$ which is a contradiction and thus finishes the proof. □

Corollary 2.3.7 *We can extend the above lemma 2.3.6 in order to include the first cycle of the cache access as well. Formally, we have* $\forall ad \in \mathbb{B}^a, t \in \mathbb{N}^+$*:*

$$valid\_input?^t \wedge \neg clear^{last_{crd}(t)} \wedge hit_{ad}^{last_{crd}(t)} \wedge ad =_s adr^{last_{crd}(t)} \implies tag^t = tag^{last_{crd}(t)}$$

Proof: Let $\neg clear^{last_{crd}(t)} \wedge hit_{ad}^{last_{crd}(t)} \wedge ad =_s adr^{last_{crd}(t)}$ hold. Since $clear^0$ holds because of $valid\_input?^t$, this leads in particular to $last_{crd}(t) > 0$. By properties 1.2.9 of *last*, we trivially conclude

$$last_{crd}(last_{crd}(t) + 1) = last_{crd}(t) \wedge \$rd^{last_{crd}(t)}.$$

Because of $\$rd^{last_{crd}(t)}$ and item 3 of $valid\_input?^t$, we conclude

$$\neg tw^{last_{crd}(t)} \wedge \neg vw^{last_{crd}(t)}$$

and thus, $tag^{last_{crd}(t)} = tag^{last_{crd}(t)+1}$ and $hit_{ad}^{last_{crd}(t)} = hit_{ad}^{last_{crd}(t)+1}$. Hence, it is sufficient to show $tag^t = tag^{last_{crd}(t)+1}$. We conclude this claim by applying lemma 2.3.6 to cycles t and $last_{crd}(t) + 1$. □

Lemma 2.3.8 *The valid part of the directory only changes on a vw or clear. Formally, we have* $\forall t \in \mathbb{N}^+, t' \in \mathbb{Z}_{\geq t}$*:*

$$\begin{aligned} &valid\_input? \wedge last_{crd}(t) = last_{crd}(t') \wedge valid^t \neq valid^{t'} \implies \\ &\quad \exists t'' \in [t : t'[: vw^{t''} \wedge val\_in^{t''} = valid^{t'}[adr^{last_{crd}(t)}[l+s-1:s]] \wedge \\ &\quad \forall l \in]t'' : t'[: \neg vw^l \end{aligned}$$

Proof: We fix an arbitrary $t \in \mathbb{N}^+$ and show the claim by induction on t'.
Induction base ($\mathbf{t' = t}$): The left-hand side of the implication is trivially false which finishes the induction base.
Induction step ($\mathbf{t' \to t'+1}$): Let $last_{crd}(t) = last_{crd}(t'+1)$ and $valid^t \neq valid^{t'+1}$ hold. We then have to find a cycle $t'' \in [t : t'+1[$ with

$$vw^{t''} \wedge val\_in^{t''} = valid^{t'}[adr^{last_{crd}(t)}[l+s-1:s]] \wedge \forall l \in]t'' : t'+1[: \neg vw^l$$

By the definition of *last*, we conclude $last_{crd}(t'+1) = last_{crd}(t')$ and $\neg crd^{t'}$, i.e., $\neg \$rd^{t'} \wedge \neg clear^{t'}$. We now split cases depending on $vw^{t'}$.

1. Let $vw^{t'}$ hold. With item 3 of $valid\_input?$, we conclude $adr^{t'} =_s adr^{last_{crd}(t)}$. We also trivially conclude

   $$val\_in^{t'} = valid^{t'+1}[adr^{last_{crd}(t)}[l+s-1:s]].$$

   Hence, cycle t' fulfills all the properties required in order to conclude the induction step.

2. Let $\neg vw^{t'}$ hold. We then have $valid^{t'} = valid^{t'+1}$ and apply the induction hypothesis in order to find a cycle $t'' \in [t : t'[$ with

   $$vw^{t''} \wedge val\_in^{t''} = valid^{t''}[adr^{last_{crd}(t)}[l+s-1:s]] \wedge \forall l \in]t'' : t'[: \neg vw^l$$

   This cycle t'' fulfills all the properties needed in order to finish the proof. □

Lemma 2.3.9 *If a byte in a cache address is written in cycle t, the cache either signals* hit^t *or the corresponding cache entry is not valid, i.e.,* $\forall b \in \mathbb{Z}_B, t \in \mathbb{N}^+$*:*

$$valid\_input? \wedge cdwb^t[b] \implies hit^t \vee \neg valid^t[adr^t[l+s-1:s]]$$

Proof: Let $cdwb^t[b]$ and $valid^t[adr^t[l+s-1:s]]$ hold. We then have to show hit^t in order to conclude the claim. Thus, it is sufficient to show $tag^t[adr^t[l+s-1:s]] = adr^t[a-1:l+s]$. By item 3 of $valid\_input?$, we conclude $adr^t =_s adr^{last_{crd}(t)} \wedge \neg clear^{last_{crd}(t)}$. We apply lemma 2.3.7 to address adr^t in order to obtain

$$hit^{last_{crd}(t)}_{adr^t} \implies tag^t = tag^{last_{crd}(t)}$$

If $hit_{adr^t}^{last_{crd}(t)}$ holds, we have

$$\begin{aligned} adr^t[a-1:l+s] &= adr^{last_{crd}(t)}[a-1:l+s] \\ &= tag^{last_{crd}(t)}[adr^t[l+s-1:s]] \\ &= tag^t[adr^t[l+s-1:s]] \end{aligned}$$

which concludes the claim. Let therefore $\neg hit_{adr^t}^{last_{crd}(t)}$ hold. Because of $adr^t =_s adr^{last_{crd}(t)}$, we also have $\neg hit^{last_{crd}(t)}$. With item 6 of $valid\_input?$, we are therefore able to find a cycle $t' \in]last_{crd}(t) : t[$ with $vw^{t'} \wedge \neg val\_in^{t'}$. By properties 1.2.9 of $last$, we then have $last_{crd}(t) = last_{crd}(t')$. With item 3 of $valid\_input?$, we conclude $adr^{t'} =_s adr^{last_{crd}(t)} =_s adr^t$. Additionally, we have $\neg valid^{t'+1}[adr^t[l+s-1:s]]$ because of $vw^{t'} \wedge \neg val\_in^{t'}$. Since $valid^t[adr^t[l+s-1:s]]$ holds, we can apply lemma 2.3.8 to cycles $t'+1$ and t in order to find a cycle $t'' \in [t'+1:t[$ with

$$vw^{t''} \wedge val\_in^{t''} \wedge \forall l \in]t'' : t[: \neg vw^l$$

By item 3 of $valid\_input?$, we once again conclude $adr^{t''} =_s adr^{last_{crd}(t'')}$ and properties 1.2.9 of $last$ lead to $last_{crd}(t'') = last_{crd}(t)$ and $\neg crd^{t''}$. We trivially conclude $hit_{adr^{t''}}^{t''+1}$ and because of $adr^t =_s adr^{t''}$, we also have

$$tag^{t''+1}[adr^t[l+s-1:s]] = adr^t[a-1:l+s]$$

We now apply lemma 2.3.6 to cycles $t''+1$ and t in order to conclude $tag^t = tag^{t''+1}$ which yields $tag^t[adr^t[l+s-1:s]] = adr^t[a-1:l+s]$ and thus concludes the claim. □

Lemma 2.3.10 *If there is a hit on some address ad in cycle t, a byte b is written in the same cycle, and ad and adr^t do not differ in line and sector part, then ad and the current address adr^t belong to the same cache line, i.e., $\forall ad \in \mathbb{B}^a, b \in \mathbb{Z}_B, t \in \mathbb{N}^+$:*

$$valid\_input? \wedge hit_{ad}^t \wedge cdwb^t[b] \wedge ad =_0^{l+s} adr^t \implies ad =_s adr^t$$

Proof: Let $hit_{ad}^t \wedge cdwb^t[b] \wedge ad =_0^{l+s} adr^t$ hold. Because of hit_{ad}^t, we conclude

$$tag^t[ad[l+s-1:s]] = ad[a-1:l+s] \wedge valid^t[ad[l+s-1:0]]$$

by equation (2.1) and definition 1.4.1 of the equal-tester. Because of $ad =_s^{l+s} adr^t$, we also have $valid^t[adr^t[l+s-1:0]]$ and we can apply lemma 2.3.9 in order to get hit^t. By the same arguments as before, this leads to

$$tag^t[ad[l+s-1:s]] = adr^t[a-1:l+s]$$

and hence, we conclude $ad =_s adr^t$. □

With these lemmas, we are finally able to prove extended cache consistency for a direct mapped cache.

Theorem 2.3.11 *Given valid_input?, a direct mapped cache fulfills the extended cache consistency predicate.*

Proof: By expanding the definition of extended cache consistency, we get the following claim:

$$\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B : hit_{ad}^t \implies \begin{array}{l} \exists_{bw(ad,b)}^{last}(t) \wedge \\ last_{bw(ad,b)}(t) > 0 \wedge \\ \left|dout_{ad}^t\right|_b = \left|din^{last_{bw(ad,b)}(t)}\right|_b \end{array}$$

Thus, given *valid_input?*, we induct on the cycle t in order to show the claim.

Induction base ($\mathbf{t = 1}$): The induction start $t = 1$ is trivial since $clear^0$ implies that in cycle 1, there can be no active *hit* signal and thus the left-hand side of the implication becomes false which concludes the induction base.

Induction step ($\mathbf{t \to t + 1}$): Let the induction hypothesis hold for cycle t. We have to show the claim for cycle $t + 1$. With lemma 2.3.3, we can conclude

$$\exists_{bw(ad,b)}^{last}(t+1) \wedge last_{bw(ad,b)}(t+1) > 0.$$

Thus, we only have to show

$$\left|dout_{ad}^{t+1}\right|_b = \left|din^{last_{bw(ad,b)}(t+1)}\right|_b.$$

By applying lemma 2.3.5, we can conclude that hit_{ad}^t holds for otherwise, this lemma would complete the induction step. We now split cases on $bw(ad, b)^t$.

1. In case $bw(ad, b)^t$ holds, we trivially conclude that

   $$last_{bw(adr,b)}(t+1) = t.$$

   By definition of $bw(ad, b)$, we conclude $ad = adr^t$ and $cdwb^t[b]$ which implies

   $$\left|dout_{adr}^{t+1}\right|_b = \left|din^t\right|_b = \left|din^{last_{bw(ad,b)}(t+1)}\right|_b$$

   by the construction of the direct mapped caches according to equations (2.1) and (2.2). This concludes the claim for this case.

2. If, on the other hand, $bw(ad, b)^t$ does not hold, we easily conclude that

   $$last_{bw(ad,b)}(t+1) = last_{bw(ad,b)}(t).$$

   Furthermore, hit_{ad}^t and the induction premise for cycle t guarantee

   $$\left|dout_{ad}^t\right|_b = \left|din^{last_{bw(ad,b)}(t)}\right|_b.$$

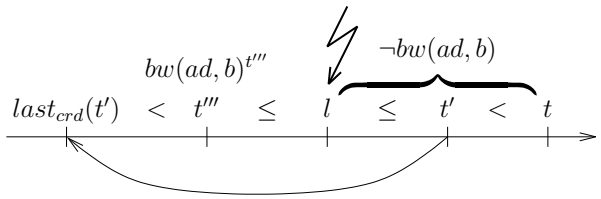

Figure 2.6: Illustration of the proof of lemma 2.3.12

Thus, it only remains to show that

$$\left|dout_{ad}^{t+1}\right|_b = \left|dout_{ad}^{t}\right|_b .$$

According to the computation of *dout* in equation (2.1), this is equivalent to

$$data^{t+1}[b][ad[l+s-1:0]] = data^{t}[b][ad[l+s-1:0]].$$

We show this claim by contradiction, i.e., we assume inequality. The implementation of the direct mapped cache according to equation (2.2) then implies that both $cdwb^t[b]$ and $ad[l+s-1:0] = adr^t[l+s-1:0]$ hold. Since in the current case, $bw(ad,b)^t$ does not hold, it is now sufficient to show $ad =_{l+s} adr^t$. Since $hit_{ad}(t)$ also holds, lemma 2.3.10 implies that $ad =_s adr^t$ which concludes this case of the induction step. □

This finishes the proof of cache consistency for the direct mapped cache. We will now show the so-called *continuous hit lemma* which plays a central role in the verification of consistency for the set-associative cache in the following section. However, we still need four auxiliary lemmas in order to show the continuous hit lemma.

Lemma 2.3.12 *If a direct mapped cache signals a hit on address ad in cycle t, then there is no line fill to the cache line of address ad between the last write to ad and the current cycle t, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$:

$$\begin{aligned} valid\_input? \wedge hit_{ad}^t \quad \Longrightarrow \quad & \forall t' \in [last_{bw(ad,b)}(t) : t[: \\ & \neg(vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'} \wedge \\ & \quad last_{crd}(t') > last_{crd}(last_{bw(ad,b)}(t))) \end{aligned}$$

Proof: Since we are going to argue about six different cycles in this proof, figure serves as an illustration. Because of hit_{ad}^t, lemma 2.3.3 guarantees

$$\exists_{bw(ad,b)}^{last}(t) \wedge last_{bw(ad,b)}(t) > 0.$$

We set $l := last_{bw(ad,b)}(t)$ and show the claim by contradiction. Assume we have a cycle $t' \in [l : t[$ with

$$vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'} \wedge last_{crd}(t') > last_{crd}(l)$$

Using property 5 of $valid\_input?$ for cycle t', we find a cycle $t'' \in]last_{crd}(t') : t'[$ with $vw^{t''} \wedge \neg val\_in^{t''}$. More importantly, for byte b of address ad, we also find a cycle $t''' \in]t'' : t']$ with $bw(ad,b)^{t'''}$. By proposition 1.2.9, we can therefore conclude $t''' \leq l$. We also know $t''' > t'' > last_{crd}(t')$ and thus,

$$last_{crd}(t''') = last_{crd}(t').$$

Furthermore, because $t''' \leq l$, we have

$$last_{crd}(t''') = last_{crd}(l),$$

i.e., we have the following equality

$$last_{crd}(t') = last_{crd}(l)$$

which is a contradiction to our initial assumption $last_{crd}(t') > last_{crd}(l)$ and thus concludes the proof. □

Lemma 2.3.13 *If a direct mapped cache signals a hit on address ad in cycle t, there was either a hit on the cycle of the last write access to ad or this last write cycle is during a line fill. Formally, we have* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$:

$$\begin{aligned} valid\_input? \wedge hit^t_{ad} \quad \Longrightarrow \quad & hit_{ad}^{last_{bw(ad,b)}(t)} \vee \\ & \exists t' \in [last_{bw(ad,b)}(t) : t[: \\ & \quad last_{crd}(t') = last_{crd}(last_{bw(ad,b)}(t)) \wedge \\ & \quad vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'} \end{aligned}$$

Proof: Because of hit^t_{ad}, lemma 2.3.3 guarantees

$$\exists^{last}_{bw(ad,b)}(t) \wedge last_{bw(ad,b)}(t) > 0.$$

We set $l := last_{bw(ad,b)}(t)$ and assume $\neg hit^l_{ad}$ for otherwise, the claim would already be concluded. With the help of lemma 2.3.2 for cycles l and t, we find a cycle $t' \in [l : t[$ with

$$vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'}.$$

We can safely assume

$$last_{crd}(t') \neq last_{crd}(l)$$

for otherwise, cycle t' would already fulfill all the properties needed in order to finish the claim. Since $t' \geq l$, proposition 1.2.9 implies $last_{crd}(t') \geq last_{crd}(l)$. Thus,

$$last_{crd}(t') > last_{crd}(l)$$

holds which is a contradiction because of lemma 2.3.12 for cycles t and t'. □

Lemma 2.3.14 *There is no active clear signal after the last write to an address that is hit, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$:

$$valid\_input? \wedge hit_{ad}^t \implies \forall t' \in [last_{bw(ad,b)}(t) : t[: \neg clear^{t'}$$

Proof: We show the claim by contradiction. Let hit_{ad}^t hold; we set $l := last_{bw(ad,b)}(t)$ and fix an arbitrary cycle $t' \in [l : t[$ with $clear^{t'}$. We trivially conclude $\neg hit_{ad}^{t'+1}$. With lemma 2.3.2 for cycles $t'+1$ and t, we find a cycle $t'' \in [t'+1 : t[$ with

$$vw^{t''} \wedge val\_in^{t''} \wedge adr^{t''} =_s ad$$

With lemma 2.3.12 for cycles t and t'', we conclude

$$last_{crd}(t'') \leq last_{crd}(l)$$

With properties 1.2.9 of $last$, we also conclude

$$last_{crd}(l) \leq last_{crd}(1+t') \leq last_{crd}(t'')$$

which leads to $last_{crd}(1+t') = last_{crd}(l)$. However, because of $clear^{t'}$, we conclude $last_{crd}(t'+1) = t'$ which is a contradiction because $l \leq t'$ and thus, $last_{crd}(l) < t'$ holds. □

Lemma 2.3.15 *There is no hit in the last cycle of a line fill, i.e.,* $\forall t \in \mathbb{N}^+$:

$$valid\_input? \wedge vw^t \wedge val\_in^t \implies \neg valid^t[adr^t[l+s-1:s]]$$

This property immediately follows from item 5 of predicate $valid\_input?$ and the proof is therefore omitted in this thesis.

Lemma 2.3.16 ***(continuous hit lemma)*** *If a direct mapped cache signals a hit in cycle t on address ad, then ad was basically hit continuously since the last write to this address, i.e., hit_{ad} held. Formally, we have* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$:

$$\begin{aligned} valid\_input? \wedge hit_{ad}^t \quad \implies \quad & \exists t' \in [last_{bw(ad,b)}(t) : t] : \\ & last_{crd}(t') = last_{crd}(last_{bw(ad,b)}(t)) \wedge \\ & \forall t'' \in [last_{bw(ad,b)}(t) : t] : hit_{ad}^{t''} = (t'' \geq t') \end{aligned}$$

Proof: Figure 2.7 illustrates the claim of the continuous hit lemma. The main idea for the proof is to construct a contradiction, i.e., if a cycle t' as in the continuous hit lemma does not exist, then there was a line fill after the last write access which is a contradiction according to lemma 2.3.12 and thus concludes the claim.

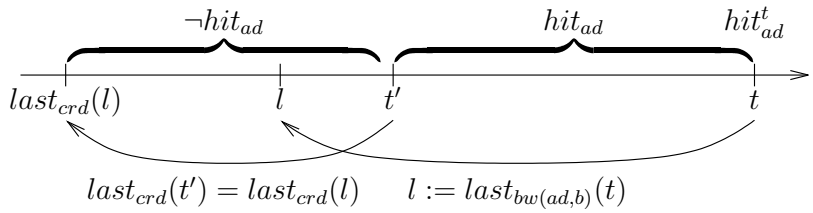

Figure 2.7: Illustration of the continuous hit lemma

Because of hit^t_{ad}, lemma 2.3.3 guarantees

$$\exists^{last}_{bw(ad,b)}(t) \wedge last_{bw(ad,b)}(t) > 0.$$

We set $l := last_{bw(ad,b)}(t)$ like in figure 2.7. By lemma 2.3.13, we can also conclude

$$hit^l_{ad} \vee \exists t' \in [l : t[: last_{crd}(t') = last_{crd}(l) \wedge vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'}$$

We now distinguish two cases:

1. The second part of the above conjunction holds, i.e., a line fill ends in cycle t'.

   $$last_{crd}(t') = last_{crd}(l) \wedge vw^{t'} \wedge val\_in^{t'} \wedge ad =_s adr^{t'}$$

   for some $t' \in [l : t[$. By using lemma 2.3.14, we can conclude $\neg clear^{t'}$. With item 3 of predicate $valid\_input?$, we additionally conclude

   $$adr^{t'} = adr^{last_{crd}(t')} \wedge \neg \$rd^{t'}.$$

   Therefore, $last_{crd}(t'+1) = last_{crd}(t')$ holds by definition. In other words,

   $$last_{crd}(t'+1) = last_{crd}(l)$$

   holds. Thus, the cycle $t'+1$ fulfills the first part of the desired property. It only remains to show

   $$hit^{t''}_{ad} = (t'' \geq t'+1)$$

   for an arbitrary $t'' \in [l : t]$. We now split cases on t''.

   (a) Let $t'' < t'+1$ hold. We then have to show $\neg hit^{t''}_{ad}$. We therefore assume $hit^{t''}_{ad}$ and find a contradiction. Since $vw^{t'} \wedge val\_in^{t'}$ holds, we know by lemma 2.3.15 that $\neg valid^{t'}[adr^{t'}[l+s-1:s]]$ holds. Because of $ad =_s adr^{t'}$ and equation (2.1) for cycle t'', we conclude

   $$\neg valid^{t'}[ad[l+s-1:s]] \wedge valid^{t''}[ad[l+s-1:s]].$$

We therefore have vw^k for some $k \in [t'' : t'[$. Item 5 of predicate $valid\_input?$ for cycle t' guarantees

$$vw^j \wedge \neg val\_in^j \wedge \exists k' \in]j : t'] : bw(ad, b)^{k'}$$

and $j = k$ because for any other $j' \in]last_{crd}(t') : t'[$, $j' \neq k$, it holds that $\neg vw^{j'}$. Thus, we have a cycle $k' \in]k : t']$ with $bw(ad, b)^{k'}$. Since $k' > k \geq t'' \geq l$, this a contradiction to the definition of $last$ and thus finishes this case of the claim.

(b) Let $t'' \geq t' + 1$ hold. We then have to show $hit_{ad}^{t''}$. We therefore assume $\neg hit_{ad}^{t''}$ and find a contradiction. With lemma 2.3.2 for cycles t'' and t, we conclude

$$vw^k \wedge val\_in^k \wedge adr^k =_s ad$$

for some $k \in [t : t''[$. Since $k \geq t > l$, we know $last_{crd}(k) \geq last_{crd}(l)$. With lemma 2.3.12 for cycles t and k, we conclude

$$\neg(vw^k \wedge val\_in^k \wedge ad =_s adr^k \wedge last_{crd}(k) > last_{crd}(l))$$

This implies that $last_{crd}(k) = last_{crd}(l) = last_{crd}(t')$. With item 5 of predicate $valid\_input?$ for cycle k, we conclude that $\neg(vw^j \wedge val\_in^j)$ holds for any $j \in [last_{crd}(k) : k[$. This is a contradiction since $vw^{t'} \wedge val\_in^{t'}$ holds and $t' \in [last_{crd}(t') : k[$.

2. The second part of the conjunction of lemma 2.3.13 does not hold. Additionally, because the whole conjunction of lemma 2.3.13 holds, hit_{ad}^{l} must hold. We show the claim for $t' := l$, i.e., because of hit_{ad}^{l}, we only have to show $hit_{ad}^{t''}$ for an arbitrary $t'' \in [l : t]$. We therefore assume $\neg hit_{ad}^{t''}$ for some t''. In this case, lemma 2.3.2 for cycles t' and t'' implies

$$vw^k \wedge val\_in^k \wedge ad =_s adr^k$$

for some $k \in [t' : t''[$. Since $k \geq t'$, we also have

$$last_{crd}(k) \geq last_{crd}(t').$$

If equality holds in the above inequality, cycle k fulfills the second part of the conjunction that in the current case is assumed to be false. We can therefore assume

$$last_{crd}(k) > last_{crd}(t').$$

Using lemma 2.3.12 for cycles t' and k, we conclude

$$\neg(vw^k \wedge val\_in^k \wedge ad =_s adr^k \wedge last_{crd}(k) > last_{crd}(t'))$$

which is a contradiction and thus finishes this case of the claim. □

For the remaining properties from definition 2.2.3, the proofs for the direct mapped cache are mostly trivial and a paper and pencil version is omitted in this thesis. In PVS, however, all these properties have been formally verified for the direct mapped cache.

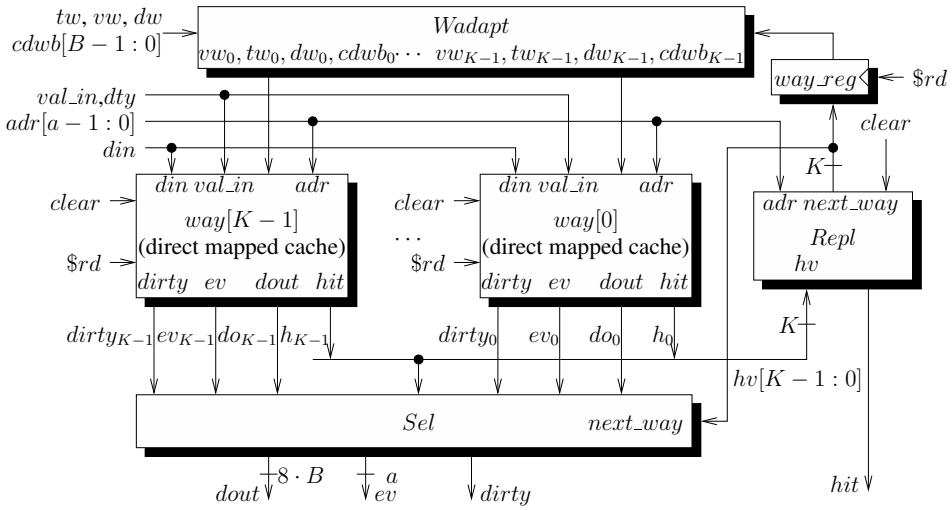

Figure 2.8: K-way set-associative cache

2.4 A set-associative cache

A K-way set-associative cache basically consists of an array of K direct mapped caches together with some glue logic and a history replacement circuit. Figure 2.8 depicts the top-level data paths of such a cache. We define $k := \lceil \log K \rceil$ for later use. The circuit *Wadapt* generates the write signals for the K direct mapped caches as follows:

$$\begin{aligned} \{vw_i, tw_i, dw_i, cdwb_i\} &= way\_reg[i] \wedge \{vw, tw, dw, cdwb\} \\ way\_reg' &= \$rd ? next\_way : way\_reg \end{aligned} \tag{2.3}$$

The circuit *Sel* computes the cache outputs. As data output *dout*, the output of the hit way is selected. For *dirty* and *ev*, however, the output of the *next way* is selected. This is due to the fact that there are *two* distinct cases where the *dirty* and *ev* outputs matter, i.e., both in case of a hit and in case of a miss. In case of a miss, *dirty* signals that the way that was selected to be evicted holds dirty data since this way is supposed to be overwritten. On the other hand, in case of a hit, *dirty* signals dirty data for the *hit* way; this behaviour is needed for invalidation caused by snooping as we will employ in our consistency protocol for the cache memory interface in chapter 3. Since the next way equals the hit way in case of a hit and the evicted way in case of a miss according to equation (2.7), we can use $next\_way$ in order to compute the cache outputs *dirty* and *ev*.

$$\begin{aligned} dout &= MUX\_us_K(do, hv) \\ dirty &= MUX\_us_K(dirty, next\_way) \\ ev &= MUX\_us_K(ev, next\_way) \end{aligned} \tag{2.4}$$

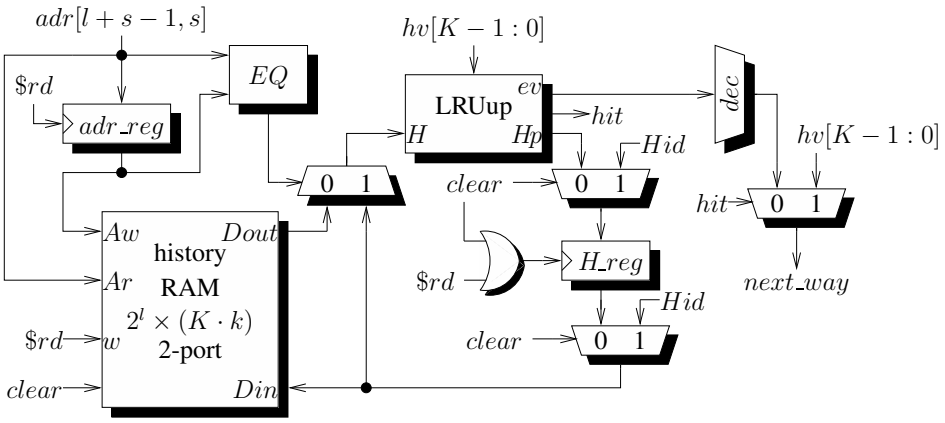

Figure 2.9: LRU replacement circuit *Repl*

The circuit *LRUrepl* in figure 2.9 basically contains a RAM for the history vectors of all cache lines and some circuits for the computation of the next history vector according to figure 2.2. Note that there are *two* pipeline stages in *LRUrepl*. The first stage reads the current history vector, updates it accordingly, and stores it in register H_reg. Additionally, the address of the cache line is stored in register adr_reg. In the second stage, the content of register H_reg is written to the location of the history RAM given by adr_reg. In particular, the content of H_reg may be forwarded to the first pipeline stage if the history access address matches the content of register adr_reg.

A history vector as introduced previously in figure 2.2 is a mapping from history positions to cache ways. In hardware, we represent this mapping by an array of size K over the domain $\mathbb{B}^k$ called *history array*. With the following definition, we establish a relationship between history vectors and history arrays.

Definition 2.4.1 *A history array H is called a* ***permutation*** *iff the function $g : \mathbb{Z}_K \to \mathbb{Z}_{2^k}$, $g(i) := \langle H[i] \rangle$ fulfills $g(\mathbb{Z}_K) = \mathbb{Z}_K$ and is a permutation. In this case, we call g the history vector* ***associated*** *with H.*

A history array H is canonically identified with a bitvector of length $K \cdot k$ by $\lambda_{i \in \mathbb{Z}_{K \cdot k}} H[i \text{ div } k][i \bmod k]$ in the schematics of the LRU circuits. We define a history array Hid that is also a permutation for initializing the history RAM, i.e.,

$$Hid := \lambda_{i \in \mathbb{Z}_K} \text{bin}_k(i). \tag{2.5}$$

Furthermore, the next way is computed in circuit *LRUrepl*. In case of a hit, the next way equals the hit way; otherwise, the least recently used way ev is used. For our formal arguments, we capture the effect of the *LRUrepl*

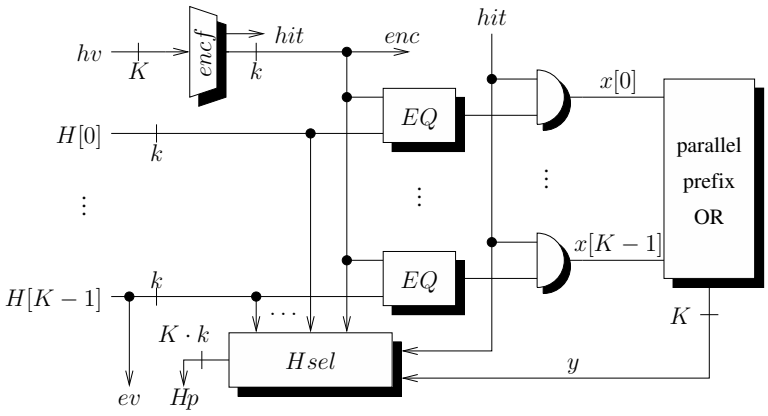

Figure 2.10: Next history computation *LRUup*

circuit into equations. We abbreviate $adr[l+s-1:s]$ by adr_l. The next configuration of the history registers according to figure 2.9 is obtained according to the following equations:

$$\begin{aligned} adr\_reg' &= \$rd?\, adr\_l : adr\_reg \\ H\_reg' &= clear?\, Hid : (\$rd?\, Hp : H\_reg) \\ history' &= \lambda_{ad\in\mathbb{B}^l} \begin{cases} Hid & \text{if } clear \\ H\_reg & \text{if } \neg clear \wedge \$rd \wedge \\ & \quad (adr\_reg = ad) \\ history[ad] & \text{otherwise} \end{cases} \end{aligned} \tag{2.6}$$

For the output of circuit *LRUrepl* in figure 2.9, the following equations hold:

$$\begin{aligned} H &= eq_l(adr\_l, adr\_reg)?\, (clear?\, Hid : H\_reg) : \\ &\quad history[adr\_l] \\ next\_way &= hit?\, hv : dec_k(ev)[K-1:0] \end{aligned} \tag{2.7}$$

$$\begin{aligned} enc &= encf_K(hv)[k-1:0] \\ hit &= encf_K(hv)[k] \\ x &= \lambda_{i\in\mathbb{Z}_K}(hit \wedge eq_k(H[i], enc)) \\ y &= PP_K(x) \\ ev &= H[K-1] \end{aligned} \tag{2.8}$$

Finally, figures 2.10 and 2.11 depict the computation of the next history vector, the hit signal, and the least recently used way ev. We also model their effect in equations. For the outputs and intermediate results of circuit *LRUup*, we have:

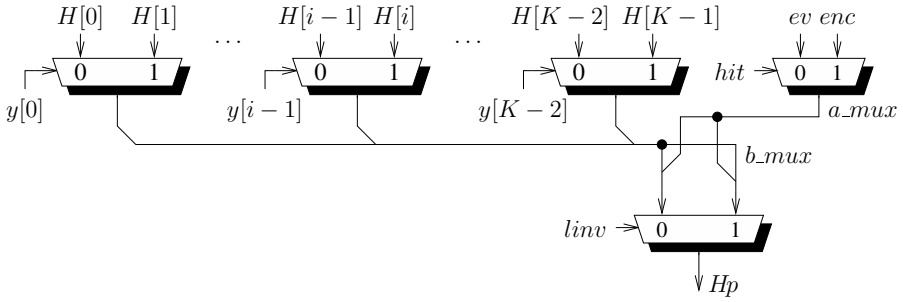

Figure 2.11: Next history computation *Hsel*

$$\begin{array}{rcl} b\_mux & = & \lambda_{i\in\mathbb{Z}_{K-1}} y[i]?\, H[i+1]{:}H[i] \\ a\_mux & = & hit?\, enc{:}ev \\ Hp & = & \begin{cases} \lambda_{i\in\mathbb{Z}_K}(i = K-1?\, a\_mux{:}b\_mux[i]) & \text{if } linv \\ \lambda_{i\in\mathbb{Z}_K}(i = 0?\, a\_mux{:}b\_mux[i-1]) & \text{otherwise} \end{cases} \end{array} \tag{2.9}$$

2.4.1 Correctness proof

Before proving anything, we introduce some notation. For the input and output signals of the direct mapped cache $way[i]$, we use hierarchical notation, e.g., $hv[i] = way[i].hit$ for the *hit* output of way i. We also use this notation for predicates, i.e., $way[i].bw(ad, b)$ denotes that predicate $bw(ad, b)$ holds for the input of way i. One important proof goal is to show that predicate $valid\_input?$ is inherited to all cache ways. Concerning the input of way i, the following equations trivially hold:

$$\begin{array}{rcl} way[i].adr & = & adr \\ way[i].\$rd & = & \$rd \\ way[i].din & = & din \\ way[i].val\_in & = & val\_in \\ way[i].dty & = & dty \\ way[i].clear & = & clear \\ way[i].\{vw, tw, dw, cdwb\} & = & \{vw, tw, dw, cdwb\} \wedge way\_reg[i] \end{array} \tag{2.10}$$

Since $way[i].\$rd = \$rd$ and $way[i].clear = clear$ both hold, we trivially have $\forall i \in \mathbb{Z}_K, t \in \mathbb{N}^+$:

$$last_{crd}(t) = last_{way[i].crd}(t) \tag{2.11}$$

Note also that the following equation holds for any way $i \in \mathbb{Z}_K$, $ad \in \mathbb{B}^a$, $b \in \mathbb{Z}_B$, and $t \in \mathbb{N}$.

$$way[i].bw(ad, b)^t \implies bw(ad, b)^t \tag{2.12}$$

Additionally, equation (2.8), lemma 1.4.8, and definition 1.4.1 guarantee

$$hit \iff \exists i \in \mathbb{Z}_K : way[i].hit. \tag{2.13}$$

Basic properties

Our first major proof goal is to show that $valid\_input?$ is inherited to all the cache ways such that we can use all the properties proved for direct mapped caches on valid input. As a next step, we will then focus on the proof that in the *way* register, at most one bit is active.

Proposition 2.4.2 *The history RAM and the registers H_reg and way_reg are only updated on a $\$rd$ or clear, i.e., $\forall t \in \mathbb{N}^+$:*

$$\begin{aligned} history^t &= history^{last_{crd}(t)+1} \\ H\_reg^t &= H\_reg^{last_{crd}(t)+1} \\ way\_reg^t &= way\_reg^{last_{crd}(t)+1} \end{aligned}$$

The proof of this proposition is omitted here due to its triviality. We now show an important lemma about the content of the way_reg register.

Proposition 2.4.3 *If the set-associative cache was hit at the beginning of the access, the current value of register way_reg is the hit vector at the beginning of the access; otherwise, at most one bit is active in the register. Formally, we have $\forall t \in \mathbb{N}$:*

$$\begin{aligned} valid\_input? \wedge \neg clear^{last_{crd}(t)} \implies\ & hit^{last_{crd}(t)}? \\ & way\_reg^t = hv^{last_{crd}(t)}: \\ & \exists j \in \mathbb{Z}_{2^k} : way\_reg^t = \lambda_{i \in \mathbb{Z}_K}(i = j) \end{aligned}$$

Proof: With proposition 2.4.2, we conclude

$$way\_reg^t = way\_reg^{last_{crd}(t)+1}$$

Because of $\neg clear^{last_{crd}(t)}$, we conclude $\$rd^{last_{crd}(t)}$. By equations (2.3) and (2.7), we conclude

$$\begin{aligned} way\_reg^{last_{crd}(t)+1} = & \ hit^{last_{crd}(t)}? \\ & hv^{last_{crd}(t)}: \\ & dec_k(ev^{last_{crd}(t)})[K-1:0] \end{aligned}$$

which finishes the claim in case $hit^{last_{crd}(t)}$ holds.

Because of the correctness of circuit *dec* from definition 1.4.1, we also know

$$dec_k(ev^{last_{crd}(t)}) = \lambda_{i \in \mathbb{Z}_{2^k}}(i = \langle ev^{last_{crd}(t)} \rangle).$$

The K lowest bits of this output are stored in the way_reg register. Thus, in case of $\neg hit^{last_{crd}(t)}$, we get with $Z_{2^k} \ni j := \langle ev^{last_{crd}(t)} \rangle$

$$way\_reg^t = \lambda_{i \in \mathbb{Z}_K}(i = j)$$

which concludes the claim. □

Note that in case $K < 2^k$, i.e., if the number of ways K is not a power of 2, the above lemma allows for the register way_reg to be empty after a miss, i.e., $way\_reg^t = 0^K$ is possible. Thus, a write to the set-associative cache in cycle t is not passed to any way; therefore, it is lost and consistency cannot be guaranteed. In order to conclude cache consistency, we will close this gap in the following section about the LRU replacement circuit and show that on a miss, exactly one bit of the register becomes active. We now proceed with the proof that $valid\_input?$ is inherited to the ways.

Lemma 2.4.4 *A cache way is only changed on an active way register, i.e., we have for any $t \in \mathbb{N}^+$ and $i \in \mathbb{Z}_K$:*

$$\neg clear^{last_{crd}(t)} \wedge \neg way\_reg^t[i] \implies way^t[i] = way^{last_{crd}(t)}[i]$$

Proof: The proof immediately follows by induction from proposition 2.4.2 and the fact that no write signals for way i are active if $way\_reg[i]$ does not hold. □

Proposition 2.4.5 *The currently active way was hit at the beginning of the cache access if and only if the whole cache was hit. Formally, we have $\forall t \in \mathbb{N}+, i \in \mathbb{Z}_K$:*

$$way\_reg^t[i] \wedge \neg clear^{last_{crd}(t)} \implies hit^{last_{crd}(t)} = way[i].hit^{last_{crd}(t)}$$

Proof: Let $way\_reg^t[i] \wedge \neg clear^{last_{crd}(t)}$ hold. With proposition 2.4.3, we conclude

$$\begin{aligned} hit^{last_{crd}(t)}? \quad & way\_reg^t = hv^{last_{crd}(t)} : \\ & \exists j \in \mathbb{Z}_{2^k} : way\_reg^t = \lambda_{i \in \mathbb{Z}_K}(i = j) \end{aligned}$$

1. In case of $hit^{last_{crd}(t)}$ holds, we have $hv^{last_{crd}(t)}[i]$ because of $way\_reg^t[i]$ which concludes this case of the claim.
2. If, on the other hand, $\neg hit^{last_{crd}(t)}[i]$ holds, we conclude $\neg hv^{last_{crd}(t)}[i]$ because of equation (2.13) which finishes the claim. □

Lemma 2.4.6 *Any way $i \in \mathbb{Z}_K$ inherits predicate $valid\_input?$ from the set-associative cache, i.e.,*

$$valid\_input? \implies way[i].valid\_input?.$$

Proof: Using equations (2.10) and (2.13), we conclude

$$\begin{aligned} way[i].hit &\implies hit \\ way[i].\{vw, tw, dw, cdwb\} &\implies \{vw, tw, dw, cdwb\}. \end{aligned}$$

Items 1 to 4 of $valid\_input?$ trivially hold with equation (2.11). We now focus on item 5 of $way[i].valid\_input?$. Let $way[i].vw^t \wedge way[i].val\_in^t$ hold. We immediately conclude $vw^t \wedge val\_in^t$ and apply item 5 of $valid\_input?$ in order to find a cycle $j \in]last_{crd}(t) : t[$ with

$$\begin{aligned} &vw^j \wedge \neg val\_in^j \wedge (\forall k \in]last_{crd}(t) : j[: \neg vw^k) \wedge \\ &(\forall k \in]j : t[: \neg vw^k) \wedge \\ &(\forall ad \in \left[adr^t\right]_s, b \in \mathbb{Z}_B : \exists t' \in]j : t] : bw(ad, b)^{t'}) \end{aligned}$$

With properties 1.2.9 of $last$, we conclude $last_{crd}(t) = last_{crd}(j)$. Applying lemma 2.4.2 twice to cycles t and j, respectively, leads to $way\_reg^t = way\_reg^j$. Since $way\_reg^t[i]$ holds, we thus have $way\_reg^j[i]$ and hence,

$$\begin{aligned} &way[i].vw^j \wedge \neg way[i].val\_in^j \wedge (\forall k \in]last_{way[i].crd}(t) : j[: \neg way[i].vw^j) \wedge \\ &(\forall k \in]j : t[: \neg way[i].vw^j) \wedge \end{aligned}$$

holds and we only have to show

$$\forall ad \in \left[way[i].adr^t\right]_s, b \in \mathbb{Z}_B : \exists t' \in]j : t] : way[i].bw(ad, b)^{t'}$$

Because of $valid\_input?$, we find a cycle $t' \in]j : t]$ with $bw(ad, b)^{t'}$. By the same arguments as above, we conclude $way\_reg^{t'}[i]$. Hence, $way[i].bw(ad, b)^{t'}$ which concludes the claim for item 5 of $way[i].valid\_input?$.

The proof of the last item basically uses the same arguments and is therefore omitted in this thesis. For details, we refer to the corresponding PVS proof. □

As a next step, we note that in case *two* or more ways of the set-associative cache signal a hit for the same address, the data output *dout* and several other circuits do not really compute anything meaningful anymore. Therefore, our next proof goal is to show *exclusiveness* for the set-associative cache, i.e., that in any cycle and for any address, at most one way signals a hit.

Closely related to exclusiveness for the set-associative cache is the register way_reg. If there is more than one active bit in way_reg on a write access, then the corresponding data is written to more than one cache way. Therefore, as a part of exclusiveness, we have a look at the way_reg register.

Definition 2.4.7 *For any $n \in \mathbb{N}^+$ a bitvector $a \in \mathbb{B}^n$ is called*

- ***unary*** *iff at most one bit in a is active, i.e.,*

$$unary?(a) := \forall i, j \in \mathbb{Z}_n : ((a[i] \wedge a[j]) \implies i = j),$$

- ***empty*** *iff no bit is active, i.e.,*

$$empty?(a) := (a = 0^n),$$

- *a* ***singleton*** *iff exactly one bit is active, i.e.,*

$$singleton?(a) := \exists i \in \mathbb{Z}_n : (a[i] \wedge \forall j \in \mathbb{Z}_n : (a[j] \implies j = i)).$$

For a singleton bitvector a, we define $the(a) \in \mathbb{Z}_n$ as the index of the active bit in a, i.e., $a[i] \iff (i = the(a))$ holds.

From this definition, we trivially conclude that a unary bitvector is either a singleton or empty. In terms of this definition, exclusiveness means that the hit vector stays unary.

Proposition 2.4.8 *Any way of the set-associative cache changes at most on some address with the same line part as the address of the cache access, i.e., we have $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$:*

$$\begin{array}{l} valid\_input? \wedge \neg clear^{last_{crd}(t)} \wedge ad \neq_s^{l+s} adr^{last_{crd}(t)} \implies \\ \quad hv_{ad}^t = hv_{ad}^{last_{crd}(t)} \wedge dout_{ad}^t = dout_{ad}^{last_{crd}(t)} \end{array}$$

Proof: We will only sketch the induction proof needed in order to show the claim. If no write signal to the cache directory or data RAM is active in the interval $[last_{crd}(t) : t[$, we trivially have $hv_{ad}^t = hv_{ad}^{last_{crd}(t)}$ and the same holds for $dout$. Hence, we find some active write signal in some cycle $t' \in [last_{crd}(t) : t[$ and item 3 of predicate $valid\_input?$ guarantees $\neg \$rd^{t'}$ and $adr^{t'} =_s adr^{last_{crd}(t')}$, i.e., in particular, $adr^{t'} =_s^{l+s} adr^{last_{crd}(t')}$. Note that proposition 1.2.9 also guarantees $last_{crd}(t') = last_{crd}(t)$.

The outputs hv_{ad} and $dout_{ad}$ of the set-associative cache are computed exclusively from the directory and data memories of the ways from address $ad[l+s-1:s]$ and $ad[l+s-1:0]$, respectively. By the above argument, the respective memories can only change on addresses different from this ad and thus, the proof is finished. □

Lemma 2.4.9 *The hit vector of a set-associative cache may only become "smaller" on an address that differs form the access address in the tag part. Formally, we have $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$:*

$$\begin{array}{l} valid\_input? \wedge \neg clear^{last_{crd}(t)} \wedge ad \neq_{l+s} adr^{last_{crd}(t)} \implies \\ \quad \forall i \in \mathbb{Z}_K : hv_{ad}^t[i] \leq hv_{ad}^{last_{crd}(t)}[i] \end{array}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): In cycle $t = 1$, all valid bits in the cache directory are invalid and thus, there is no hit which concludes the induction base.

Induction step ($\mathbf{t \to t+1}$): Let $\neg clear^{last_{crd}(t+1)} \wedge ad \neq_{l+s} adr^{last_{crd}(t+1)}$ hold. We fix an arbitrary $i \in \mathbb{Z}_K$ and assume $hv_{ad}^{t+1}[i]$ because otherwise, there is nothing to show. We then have to prove $hv_{ad}^{last_{crd}(t+1)}[i]$. This is equivalent to assuming $way[i].hit_{ad}^{t+1}$ and $\neg way[i].hit_{ad}^{last_{crd}(t+1)}$ an finding a contradiction. We split cases on crd^t.

1. Let crd^t hold. We then have $last_{crd}(t+1) = t$. Lemma 2.3.1 for way i then guarantees $way[i].tw^t \wedge way[i].adr^t =_s ad$. Because of $way[i].adr^t = adr^t$, this is a contradiction to $ad \neq_{l+s} adr^t$ and thus finishes this case.

2. Let $\neg crd^t$ hold. Thus, we conclude $last_{crd}(t+1) = last_{crd}(t)$. With the induction hypothesis, we conclude $way[i].hit_{ad}^t \leq way[i].hit_{ad}^{last_{crd}(t)}$. Thus, we can assume $\neg way[i].hit_{ad}^t$ because otherwise, the claim is already concluded. With lemma 2.3.1 for way i, we conclude $way[i].tw^t \wedge way[i].adr^t =_s ad$. Because of $valid\_input?$ and tw^t, we conclude $adr^t =_s adr_{last_{crd}(t)}$. This leads to $adr_{last_{crd}(t+1)} =_s ad$ which is a contradiction to $ad \neq_{l+s} adr^t$ and thus concludes the claim. □

Lemma 2.4.10 *A hit vector in the set-associative cache is "smaller" than the register way_reg; formally, for $t \in \mathbb{N}^+, ad \in \mathbb{B}^a$:*

$$valid\_input?^t \wedge \neg clear^t \wedge ad =_s adr^{last_{crd}(t)} \implies \forall i \in \mathbb{Z}_K : hv_{ad}^t[i] \leq way\_reg^{last_{crd}(t)+1}[i]$$

Proof: Let $valid\_input?^t \wedge \neg clear^t \wedge ad =_s adr^t$ hold. We fix an arbitrary $i \in \mathbb{Z}_K$ and have to show $hv_{ad}^t[i] \leq way\_reg^{last_{crd}(t)+1}[i]$. Let therefore $hv_{ad}^t[i]$ hold because otherwise, there is nothing to show. We will show the claim by contradiction, i.e., we assume $\neg way\_reg^{last_{crd}(t)+1}[i]$.

With proposition 2.4.2, we conclude $\neg way\_reg^t[i]$. With lemma 2.4.4, we get $way^t[i] = way^{last_{crd}(t)}[i]$; in particular, this means $hv_{ad}^{last_{crd}(t)}[i]$ and thus $hit_{ad}^{last_{crd}(t)}$. Because $ad =_s adr^{last_{crd}(t)}$, this also implies $hit^{last_{crd}(t)}$. We now apply proposition 2.4.3 in order to conclude

$$way\_reg^t = hv^{last_{crd}(t)}$$

This leads to $way\_reg^t[i]$ which is a contradiction and thus concludes the claim. □

Lemma 2.4.11 *The hit vector of the set-associative cache stays unary for addresses on a line different from the line of the address of the current cache access. Formally, we have $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$:*

$$valid\_input?^t \wedge \neg clear^{last_{crd}(t)} \wedge ad \neq_s adr^{last_{crd}(t)} \wedge unary?(hv_{ad}^{last_{crd}(t)}) \implies unary?(hv_{ad}^t)$$

Proof: Let $\neg clear^{last_{crd}(t)} \wedge ad \neq_s adr^{last_{crd}(t)}$ and $unary?(hv_{ad}^{last_{crd}(t)})$ hold. We have to show $unary?(hv_{ad}^{t})$. With proposition 2.4.8, we conclude that $ad =_s^{l+s} adr^{last_{crd}(t)}$ because otherwise, $hv_{ad}^{t} = hv_{ad}^{last_{crd}(t)}$ would conclude the claim. Since $ad \neq_s adr^{last_{crd}(t)}$ holds, this leads to $ad \neq_{l+s} adr^{last_{crd}(t)}$. Thus, we can apply lemma 2.4.9 in order to conclude $\forall i \in \mathbb{Z}_K : hv_{ad}^{t}[i] \leq hv_{ad}^{last_{crd}(t)}[i]$ which concludes the claim. □

Lemma 2.4.12 *The hit vector of the set-associative cache stays unary for addresses in the same line as the address of the current cache access. Formally, we have* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$*:*

$$valid\_input? \wedge \neg clear^{last_{crd}(t)} \wedge ad =_s adr^{last_{crd}(t)} \wedge unary?(hv_{ad}^{last_{crd}(t)}) \implies unary?(hv_{ad}^{t})$$

Proof: Let $\neg clear^{last_{crd}(t)} \wedge ad =_s adr^{last_{crd}(t)} \wedge unary?(hv_{ad}^{last_{crd}(t)})$ hold. We have to show $unary?(hv_{ad}^{t})$. By lemma 2.4.10, we know that

$$\forall i \in \mathbb{Z}_K : hv_{ad}^{t}[i] \leq way\_reg^{last_{crd}(t)+1}[i].$$

By definition 2.4.7, it is hence sufficient to show $unary?(way\_reg^{last_{crd}(t)+1})$. With lemma 2.4.2, we additionally get $way\_reg^{last_{crd}(t)+1} = way\_reg^{t}$. With proposition 2.4.3, we conclude that either $way\_reg^{t} = hv^{last_{crd}(t)} = hv_{ad}^{last_{crd}(t)}$ which concludes the claim or we have $\exists j \in \mathbb{Z}_{2^k} : way\_reg^{t} = \lambda_{i \in \mathbb{Z}_K}(i = j)$ which trivially leads to $unary?(way\_reg^{t})$ and thus finishes the claim. □

Theorem 2.4.13 *The hit vector in the set-associative cache stays unary. Formally, we have for any* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$*:*

$$valid\_input? \implies unary?(hv_{ad}^{t})$$

Proof: The proof follows immediately by induction from lemmas 2.4.11 and 2.4.12 and the trivial property that all initial hit vectors are unary because they are empty after a *clear*. □

LRU replacement circuits

As a next step, we show some essential properties of the LRU replacement circuit that are needed in order to prove cache consistency for the set-associative cache. Note that in case all the bits in the *way* register are cleared any write to the set-associative cache is lost since it does not access any cache way.

We start by formalizing the specification of the next history vector as depicted in figure 2.2 on page 27. Let h be a history vector, i.e., h is a permutation on $\mathbb{Z}_K$. In case of a hit, i.e., $singleton?(hv)$, let $i := the(hv)$

hold. Depending on *hit* and *linv*, we then define a function $h' : \mathbb{Z}_K \to \mathbb{Z}_K$ by the following equations.

$$\begin{aligned}
hit \wedge linv &: h' = \lambda_{j\in\mathbb{Z}_K} \begin{cases} i & \text{if } j = K-1 \\ h(j) & \text{if } j < h^{-1}(i) \\ h(j+1) & \text{otherwise} \end{cases} \\
hit \wedge \neg linv &: h' = \lambda_{j\in\mathbb{Z}_K} \begin{cases} i & \text{if } j = 0 \\ h(j) & \text{if } j > h^{-1}(i) \\ h(j-1) & \text{otherwise} \end{cases} \qquad (2.14) \\
\neg hit \wedge linv &: h' = h \\
\neg hit \wedge \neg linv &: h' = \lambda_{j\in\mathbb{Z}_K} \begin{cases} h(K-1) & \text{if } j = 0 \\ h(j-1) & \text{otherwise} \end{cases}
\end{aligned}$$

Note that h' as defined above is still a permutation if h is a permutation which we will not formally prove in this thesis since the claim is straightforward. Let the history array H be a permutation and the hit vector hv be unary. We now want to show that *Hp* as computed according to equations (2.8) and (2.9) is a permutation. In particular, if h is the history vector associated with H, then h' according to the above definition is associated with *Hp*.

Lemma 2.4.14 *The input x of the parallel prefix or circuit is correct, i.e.,*

$$x = \begin{cases} 0^K & \textit{if } \neg hit \\ \lambda_{j\in\mathbb{Z}_K}(j = h^{-1}(the(hv))) & \textit{otherwise} \end{cases}$$

Proof: According to equation (2.8) and definition 1.4.1 for the equality tester, we have

$$x = \lambda_{j\in\mathbb{Z}_K}(hit \wedge (H[j] = encf_K(hv)[K-1:0])).$$

In case $\neg hit$ holds, we trivially have $x = 0^K$ which concludes the claim. Let therefore *hit* hold. Since hv is unary and *hit* holds, it is also a singleton and lemma 1.4.8 guarantees $encf_K(hv)[k-1:0] = \mathrm{bin}_k(the(hv))$. We therefore have

$$x = \lambda_{j\in\mathbb{Z}_K}(H[j] = \mathrm{bin}_k(the(hv))).$$

Since H is a permutation, we conclude with the definition of the associated history vector

$$H[j] = \mathrm{bin}_k(the(hv)) \iff h(j) = the(hv).$$

Additionally, because H is a permutation, we get

$$x = \lambda_{j\in\mathbb{Z}_K}(j = h^{-1}(the(hv)))$$

which concludes the claim. □

Lemma 2.4.15 *The output y of the parallel prefix or circuit is correct, i.e.,*

$$y = \begin{cases} 0^K & \textit{if } \neg hit \\ \lambda_{j \in \mathbb{Z}_K}(j \geq h^{-1}(the(hv))) & \textit{otherwise} \end{cases}$$

Proof: According to lemma 2.4.14, the input x of the parallel prefix or is given by

$$x = \begin{cases} 0^K & \text{if } \neg hit \\ \lambda_{j \in \mathbb{Z}_K}(j = h^{-1}(the(hv))) & \text{otherwise} \end{cases}$$

The correctness of the parallel prefix or from lemma 1.4.6 additionally guarantees

$$y = \lambda_{j \in \mathbb{Z}_K} \bigvee_{l=0}^{l \leq j} x[l].$$

By replacing x, we conclude

$$y = \begin{cases} \lambda_{j \in \mathbb{Z}_K} \bigvee_{l=0}^{l \leq j} 0 & \text{if } \neg hit \\ \lambda_{j \in \mathbb{Z}_K} \bigvee_{l=0}^{l \leq j} (l = h^{-1}(the(hv))) & \text{otherwise} \end{cases}$$

This simplifies to

$$y = \begin{cases} 0^K & \text{if } \neg hit \\ \lambda_{i \in \mathbb{Z}_K}(j \geq h^{-1}(the(hv))) & \text{otherwise} \end{cases}$$

and therefore the proof is finished. □

Lemma 2.4.16 *The intermediate signal a_mux is correct, i.e.,*

$$\langle a\_mux \rangle = hit ?\, the(hv) : h(K-1)$$

Proof: According to equation (2.9) and (2.8), we have

$$a\_mux = hit ?\, encf_K(hv)[k-1:0] : H[K-1].$$

With the correctness of $encf_K(hv)[k-1:0]$ from the intermediate results of lemma 2.4.14, we conclude $encf_K(hv)[K-1:0] = bin_K(the(hv))$. The proof of the lemma follows immediately because h is associated to H. □

Lemma 2.4.17 *The intermediate signal b_mux is correct, i.e., $\forall j \in \mathbb{Z}_{K-1}$:*

$$\langle b\_mux[j] \rangle = (hit \wedge j \geq h^{-1}(the(hv))) ?\, h(j+1) : h(j)$$

Proof: According to equation (2.9), we have

$$b\_mux = \lambda_{j \in \mathbb{Z}_{K-1}} y[j] ?\, H[j+1] : H[j].$$

Since h is associated with H, this is equivalent to

$$\langle b\_mux \rangle = \lambda_{j \in \mathbb{Z}_{K-1}} y[j] ?\, h[j+1] : h[j].$$

Because of lemma 2.4.15, we can replace $y[j]$ by $hit ?\, j \geq h^{-1}(the(hv)) : 0$ which equals $hit \wedge j \geq h^{-1}(the(hv))$ and therefore, the proof is finished. □

Now, we are finally able to conclude the correctness of the next history circuit.

Theorem 2.4.18 *Let H be a permutation, h its associated history vector, and hv a unary hit vector. The next history vector h' according to equation (2.14) is then associated to Hp as computed by the hardware.*

Proof: According to equation (2.9), Hp is computed as follows:

$$Hp = \begin{cases} \lambda_{j \in \mathbb{Z}_K}(j = 0 ?\, a\_mux : b\_mux[j-1]) & \text{if } linv \\ \lambda_{j \in \mathbb{Z}_K}(j = K-1 ?\, a\_mux : b\_mux[j]) & \text{otherwise} \end{cases}$$

We split cases on the values of $linv$ and hit. Since the proofs of the different cases are very similar, we only focus on one of the cases here. The other cases, however, are also proved in PVS.

Let $hit \wedge \neg linv$ hold. With lemma 2.4.16 and 2.4.17, we then have

$$\lambda_{j \in \mathbb{Z}_K} \langle Hp[j] \rangle = \lambda_{j \in \mathbb{Z}_K} \begin{cases} the(hv) & \text{if } j = 0 \\ h(j) & \text{if } j - 1 \geq h^{-1}(i) \\ h(j-1) & \text{otherwise} \end{cases}$$

which concludes this case of the claim given the specification from equation (2.14). □

In particular, we have shown that the next history vector stays a permutation given a unary hit vector. Combined with the results about exclusiveness, we can therefore conclude that at any cycle t, the history RAM only contains permutations.

Theorem 2.4.19 *The content of the history RAM only contains permutations. Formally, since we also support forwarding, we claim the output H of the forwarding multiplexer of figure 2.9 on page 47 to be a permutation, i.e., we have $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^l$:*

$$valid\_input ? \implies permutation ?(fw_{ad}^t)$$

The proof of this claim follows trivially from theorems 2.4.13 and 2.4.18 as well as the fact that the initial history vector Hid is also a permutation. We now list a few corollaries without explicit proof.

Corollary 2.4.20 *On a hit, the hit vector of a set-associative cache is a singleton, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$*:*

$$valid\_input?^t \wedge hit_{ad}^t \implies singleton?(hv_{ad}^t)$$

Corollary 2.4.21 *The way register of a set-associative cache is correct, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned} &valid\_input?^t \wedge \neg clear^{last_{crd}(t)} \implies \\ &\quad way\_reg^t = hit^{last_{crd}(t)}?\, hv^{last_{crd}(t)} : \lambda_{i \in \mathbb{Z}_K}(i = \langle ev^{last_{crd}(t)} \rangle) \end{aligned}$$

Corollary 2.4.22 *The content of the register* way_reg *always has one active bit, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$valid\_input?^t \wedge \neg clear^{last_{crd}(t)} \implies singleton?(way\_reg^t)$$

Lemma 2.4.23 *In case of a hit, the hit vector equals the register* way_reg *in the cycle* after *the beginning of the access, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a$*:*

$$valid\_input?^t \wedge ad =_s adr^{last_{crd}(t)} \wedge hit_{ad}^t \implies way\_reg^{last_{crd}(t)+1} = hv_{ad}^t$$

Proof: We set $l := last_{crd}(t)$. Let $ad =_s adr^l \wedge hit_{ad}^t$ hold. We have to show $way\_reg^{l+1} = hv_{ad}^t$. We therefore fix an arbitrary $i \in \mathbb{Z}_K$ and show $way\_reg^{l+1}[i] = hv_{ad}^t[i]$. With corollary 2.4.20, we conclude $singleton?(hv_{ad}^t)$. Since hit_{ad}^t holds, we also have $\neg clear^l$. With proposition 2.4.2, we conclude $way\_reg^t = way\_reg^{l+1}$. Additionally, we get $singleton?(way\_reg^{l+1})$ by corollary 2.4.22. Lemma 2.4.10 further guarantees

$$\forall j \in \mathbb{Z}_K : hv_{ad}^t[j] \leq way\_reg^{l+1}[j]$$

For $j = the(hv_{ad}^t)$, this leads to $way\_reg^{l+1}[j]$. The claim $way\_reg^{l+1} = hv_{ad}^t$ then follows by definition 2.4.7 of $singleton?$. □

Cache consistency

The main proof idea for cache consistency of the set-associative cache is to reduce it to cache consistency for the ways, i.e., the direct mapped caches. Cache consistency for the cache ways already guarantees that the data output on a hit is the last data written *to the hit way*. If this is also the last data written to the *whole* cache, we have concluded cache consistency for the set-associative cache. The continuous hit lemma 2.3.16 will play a crucial part in this proof. We therefore extend the continuous hit lemma to the set-associative cache.

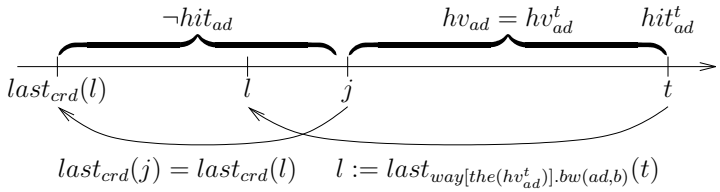

Figure 2.12: Illustration of the claim of lemma 2.4.24

Lemma 2.4.24 *In a set-associative cache, a* ***continuous hit lemma*** *similar to lemma 2.3.16 for direct mapped caches holds. We illustrate the claim in figure 2.12. Formally, we have* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$*:*

$$\begin{aligned} &valid\_input?^t \wedge hit^t_{ad} \implies \exists j \in [last_{way[the(hv^t_{ad})].bw(ad,b)}(t) : t] : \\ &\quad last_{crd}(j) = last_{crd}(last_{way[the(hv^t_{ad})].bw(ad,b)}(t)) \wedge \\ &\quad \forall k \in [last_{way[the(hv^t_{ad})].bw(ad,b)}(t) : t] : k \geq j ?\, hv^t_{ad} = hv^k_{ad} : \neg hit^k_{ad} \end{aligned}$$

Proof: Let hit^t_{ad} hold. Lemma 2.4.20 then ensures $singleton?(hv^t_{ad})$. With lemma 2.4.6 for way $i := the(hv^t_{ad})$, we conclude $way[i].valid\_input?^t$. Because of $way[i].hit^t_{ad}$, we additionally get $\exists^{last}_{way[i].bw(ad,b)}(t)$ by lemma 2.3.3 and set $l := last_{way[i].bw(ad,b)}(t)$. The continuous hit lemma 2.3.16 for way i leads to a cycle $t' \in [: t]$ with

$$\begin{aligned} &last_{way[i].crd}(t') = last_{way[i].crd}(l) \wedge \\ &\forall t'' \in [l : t] : way[i].hit^{t''}_{ad} = (t'' \geq t') \end{aligned} \tag{2.15}$$

With equation (2.11), we conclude $last_{crd}(t') = last_{crd}(l)$. Thus, we only have to show

$$t'' \geq t' ?\, hv^t_{ad} = hv^{t''}_{ad} : \neg hit^{t''}_{ad}$$

for an arbitrary $t'' \in [l : t]$. We therefore fix such a cycle t'' and instantiate equation (2.15) with it in order to get $way[i].hit^{t''}_{ad} = (t'' \geq t')$. If $hit^{t''}_{ad}$ does not hold, $way[i].hit^{t''}_{ad}$ cannot hold either which leads to $t'' < t$ and concludes the claim. Let therefore $hit^{t''}_{ad}$ hold. We split cases on $t'' \geq t'$.

1. Let $t'' \geq t'$ hold. We then know $way[i].hit^{t''}_{ad}$ and have to conclude $hv^t_{ad} = hv^{t''}_{ad}$. With lemma 2.4.20, we conclude $singleton?(hv^{t''}_{ad})$. Since $the(hv^{t''}_{ad}) = the(hv^t_{ad}) = i$ holds because of $way[i].hit^{t''}_{ad}$, this case of the claim is trivially concluded.

2. Let $t'' < t'$ hold. We then know $\neg way[i].hit^{t''}_{ad}$ and can assume $hit^{t''}_{ad}$. We will show that this is a contradiction. We instantiate equation (2.15) with cycle t' and get $way[i].hit^{t'}_{ad}$. With properties 1.2.9 of $last$, we conclude $last_{crd}(t'') = last_{crd}(t')$. Additionally, $way[i].bw(ad,b)^l$ holds by definition; in particular, $cdwb^l[b]$ and $ad = adr^l$ both hold. With item 3 of $valid\_input?^t$ for cycle l, this leads to

$$ad =_s adr^{last_{crd}(l)}$$

and thus, we get $ad =_s adr^{last_{crd}(t')} = adr^{last_{crd}(t'')}$. We now apply lemma 2.4.23 for both cycles t' and t'' in order to get $hv_{ad}^{t''} = hv_{ad}^{t'}$. This immediately leads to $way[i].hit_{ad}^{t''}$ which is a contradiction and thus finishes the claim. □

Lemma 2.4.25 *If a set-associative cache signals a hit, no byte in the hit address is written in the whole cache after the last write to the hit way, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$*:*

$$valid\_input? \wedge hit_{ad}^t \implies \forall t' \in]last_{way[the(hv_{ad}^t)].bw(ad,b)}(t) : t[: \neg bw(ad,b)^{t'}$$

Proof: Let hit_{ad}^t hold. According to lemma 2.4.20, $singleton?(hv_{ad}^t)$ holds and thus, $the(hv_{ad}^t)$ is defined. We introduce the shorthand notation $i := the(hv_{ad}^t)$. In particular, $way[i].valid\_input?$ holds according to lemma 2.4.6; because of $way[i].hit_{ad}^t$, we additionally get $\exists^{last}_{way[i].bw(ad,b)}(t)$ by lemma 2.3.3 and once again set $l := last_{way[i].bw(ad,b)}(t)$. We fix an arbitrary $t' \in]l : t[$ and show the claim by contradiction, i.e., we assume $bw(ad,b)^{t'}$ holds. By the definition of *last*, we conclude $\neg way[i].bw(ad,b)^{t'}$, i.e., $\neg way\_reg^{t'}[i]$. With lemma 2.4.24, we find a cycle $j \in [l : t]$ with

$$\begin{array}{l} last_{crd}(j) = last_{crd}(l) \wedge \\ \forall k \in [l : t] : k \geq j ? hv_{ad}^t = hv_{ad}^k : \neg hit_{ad}^k \end{array} \tag{2.16}$$

We now split cases on whether cycle t' was in the same cache access as the last write to the hit way.

1. Let $last_{crd}(t') = last_{crd}(l)$ hold. By applying proposition 2.4.2 twice for cycles t' and j, we get $way\_reg^{t'} = way\_reg^j$. We instantiate equation (2.16) with cycle j in order to get $hv_{ad}^t = hv_{ad}^j$. If $ad =_s adr^{last_{crd}(j)}$ holds, lemma 2.4.23 ensures $way\_reg^j[i]$ which is a contradiction. Let therefore $ad \neq_s adr^{last_{crd}(j)} = adr^{last_{crd}(t')}$ hold. However, we know $ad = adr^{t'}$ and $cdwb^{t'}[b]$ because of $bw(ad,b)^{t'}$. Item 3 of $valid\_input?$ then ensures that $ad =_s adr^{last_{crd}(t')}$ which is a contradiction and thus finishes this case of the claim.

2. Let $last_{crd}(t') \neq last_{crd}(l)$ hold. Hence, $last_{crd}(j) \neq last_{crd}(t')$ also holds. We conclude with properties 1.2.9 of *last* that $t' \notin]last_{crd}(j) : j]$ and
$$t' > l > last_{crd}(l) = last_{crd}(j).$$
Thus, we get $t' > j$. We can now instantiate equation (2.16) with cycle t' in order to get $hv_{ad}^t = hv_{ad}^{t'}$. Proposition 2.4.2 for cycle t' guarantees $way\_reg^{t'} = way\_reg^{last_{crd}(t')+1}$. Additionally, item 3 of $valid\_input?$ yields $adr^{t'} =_s adr^{last_{crd}(t')}$ because of $bw(ad,b)^{t'}$. With lemma 2.4.23 for cycle t', we then conclude $way\_reg^{last_{crd}(t')+1} = hv_{ad}^{t'}$

and hence, $way\_reg^{t'}[i]$ which is a contradiction and thus finishes the claim. □

Corollary 2.4.26 *If a set-associative cache signals a hit, the cycle of the last write to the set-associative cache of a byte in the hit address equals the cycle of the last write to the hit cache way, i.e.,* $\forall t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$*:*

$$valid\_input? \wedge hit_{ad}^t \implies last_{way[the(hv_{ad}^t)].bw(ad,b)}(t) = last_{bw(ad,b)}(t)$$

Proof: The proof of this corollary immediately follows from lemma 2.4.25, the definition of *last*, and equation (2.12). □

Now, we are finally able to conclude cache consistency for the K-way set-associative cache.

Theorem 2.4.27 *Given valid input, any set-associative cache fulfills the extended cache consistency predicate.*

Proof: Since hit_{ad}^t holds, we conclude $singleton?(hit_{ad}^t)$ with theorem 2.4.13, i.e., there exists exactly one way $i := the(hv_{ad}^t)$ with $way[i].hit_{ad}^t$ and we also have $way[i].dout_{ad}^t = dout_{ad}^t$. Because of lemma 2.4.6, $way[i].valid\_input?$ holds. We instantiate extended cache consistency from theorem 2.3.11 for this way i and thus get

$$\exists_{way[i].bw(ad,b)}^{last}(t) \wedge last_{way[i].bw(ad,b)}(t) > 0 \wedge \left|way[i].dout_{ad}^t\right|_b = \left|way[i].din^l\right|_b$$

We set $l := last_{way[i].bw(ad,b)}(t)$. Because of corollary 2.4.26, we conclude $l = last_{bw(ad,b)}(t)$, i.e., in particular,

$$\exists_{bw(ad,b)}^{last}(t) \wedge last_{bw(ad,b)}(t) > 0$$

holds. Because of

$$\left|dout_{ad}^t\right|_b = \left|way[i].dout_{ad}^t\right|_b = \left|din^l\right|_b = \left|din^{last_{bw(ad,b)}(t)}\right|_b,$$

the proof of extended cache consistency for the set-associative cache is finished. □

2.5 A fully associative cache

A fully associative cache basically is just a set-associative cache with $l = 0$, i.e., each cache way just holds *one* cache line. For simplicity, we assume that the number of ways K is a power of 2, i.e., $k = \log K$ holds with k defined as in the previous section. Since each cache way only contains one line, the directory of each way contains one entry. The directory entries to K RAMs

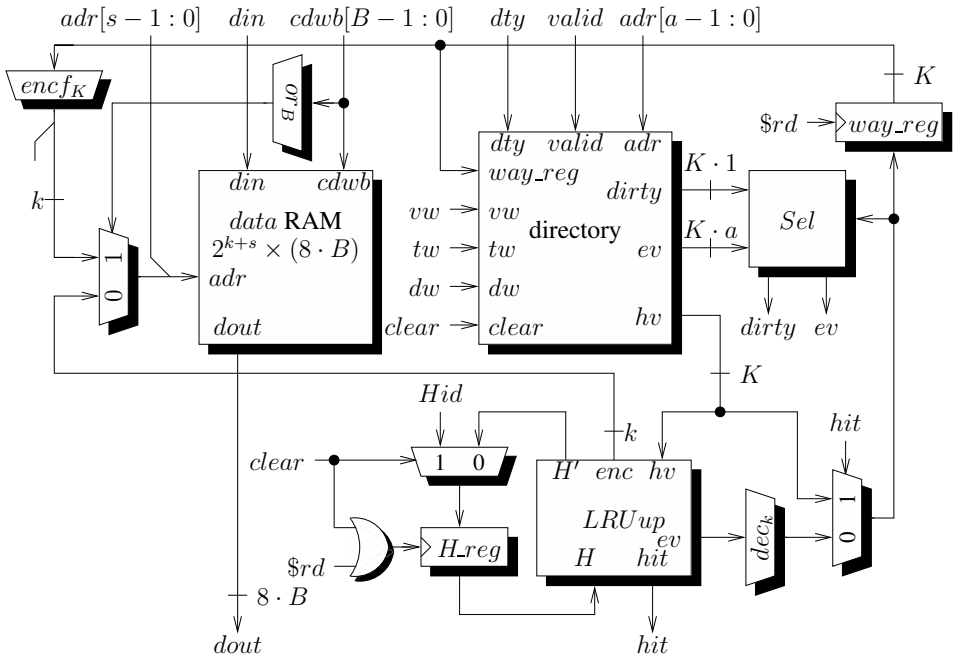

Figure 2.13: Fully associative cache

with one entry each are instead combined into an *array* of registers. They cannot be combined into a RAM since they all have to be accessed in parallel in order to compute the *hit* signal.

For the data RAMs of the different ways, a similar approach is chosen. In the set-associative cache with $l = 0$, they only contain one cache line each, i.e., 2^s data words. Since they do not have to be accessed in parallel, we can combine them into *one* data RAM that contains 2^{k+s} data words. However, we then have *two* different access addresses for the data RAM depending on the type of an access. During a write, i.e., if $cdwb[b]$ holds for some b, the encoded way_reg register is used for the upper k bits of the access address; otherwise, we use the encoded hit vector. The lower part of both addresses is formed by the sector part of the original address. Thus, we need a $2^{k+s} \times (8 \cdot B)$ data RAM as depicted in figure 2.13.

The history RAM only contains one history vector and thus, it is implemented with a single register. Note that a second pipeline stage for the history computation with forwarding as in the set-associative cache is not necessary. Figure 2.13 depicts the top-level data paths for this fully associative cache. The implementation of the cache directory is shown in figure 2.14. Note that the environments *Sel* and *LRUup* have already been introduced for the set-associative cache in equation (2.4) and figure 2.9. We now introduce the corresponding equations that differ from the set-associative cache with $l = 0$ for our formal arguments:

$$radr = (or_B(cdwb)?\, encf_K(way\_reg)[k-1:0]:enc) \cdot adr[s-1:0] \quad (2.17)$$

$$dout = \lambda_{i\in\mathbb{Z}_{8\cdot B}} data[i \text{ div } 8]\,[radr]\,[i \bmod 8] \quad (2.18)$$

$$data' = \lambda_{b\in\mathbb{Z}_B}\lambda_{x\in\mathbb{B}^{k+s}} \begin{cases} |din|_b & \text{if } cdwb[b] \wedge x = radr \\ data[b][x] & \text{otherwise} \end{cases} \quad (2.19)$$

2.5.1 Correctness proof

The main idea of the correctness proof for the fully associative cache is to reduce it to the correctness of a set-associative cache with the same set of parameters verified in section 2.4.1.

Definition 2.5.1 *A fully associative cache fa and a set-associative cache sa are called equivalent, i.e., $sa \equiv fa$, iff the following conditions are fulfilled $\forall i \in \mathbb{Z}_K$:*

$$\begin{aligned} sa.way[i].data &= \lambda_{b\in\mathbb{Z}_B}\lambda_{x\in\mathbb{B}^s} fa.data[b][\mathrm{bin}_k(i)\cdot x] \wedge \\ sa.way[i].dir &= \lambda_{x\in\mathbb{B}^0} fa.dir[i] \wedge \\ sa.way\_reg &= fa.way\_reg \wedge \\ sa.H\_reg &= fa.H\_reg \end{aligned}$$

For any fully associative cache fa, we define a function $sa(fa)$ that maps it to an equivalent set-associative cache, i.e., $sa(fa) \equiv fa$ holds by construction. The definition of the function $sa(fa)$ is trivially derived from the above definition of the equivalence. For the history RAM and the address register of the set-associative cache which are not part of the above equivalence, we define:

$$\begin{aligned} sa(fa).adr\_reg &:= 0^0 \\ sa(fa).history &:= \lambda_{x\in\mathbb{B}^0} Hid \end{aligned}$$

The following proposition basically states that equivalent caches produce the same output.

Proposition 2.5.2 *Let sa be a configuration of a set-associative cache, fa a configuration of a fully associative cache, let and let $fa \equiv sa$ hold. The outputs of sa and fa are then equal. In case of dout, we need the additional assumption that the hit vector is actually a singleton, i.e., there is exactly one hit way, and that in case of a byte write in cycle t, the hit vector equals the*

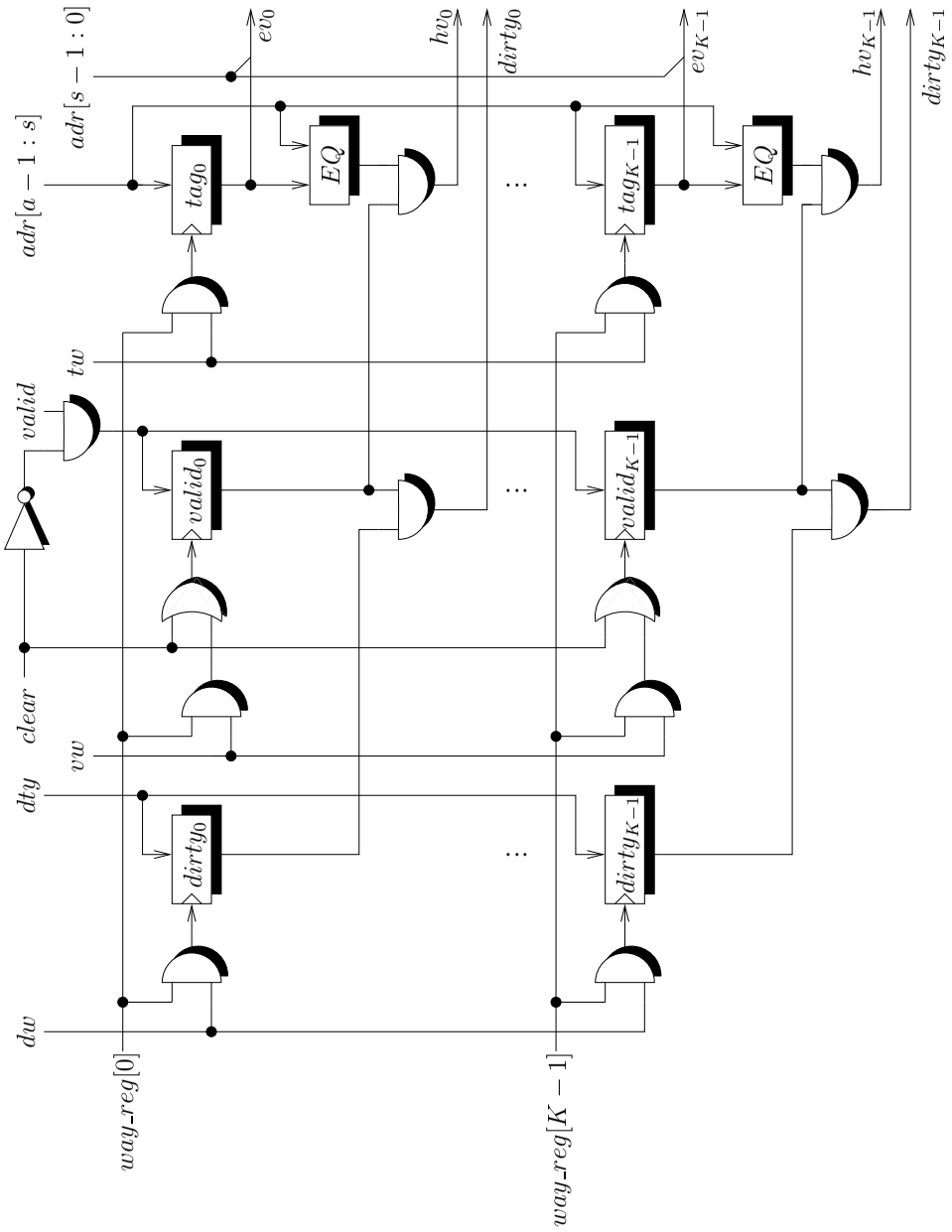

Figure 2.14: Directory environment of a fully associative cache

way_reg register. Later on, we will justify these assumptions with lemmas proved about the set-associative cache. Formally, we then have $\forall ad \in \mathbb{B}^a$*:*

$$\begin{array}{l} sa.hv = fa.hv \wedge sa.dirty = fa.dirty \wedge sa.ev = fa.ev \wedge \\ (fa.hit \wedge singleton?(fa.hv) \wedge \\ \quad (cdwb^t \neq 0^B \implies fa.way\_reg = fa.hv) \implies \\ \quad\quad sa.dout = fa.dout) \end{array}$$

Proof: The claims about the hit vector hv as well as the outputs $dirty$ and ev follow immediately form the construction of the fully associative cache and definition 2.5.1 of equivalence. We therefore only focus on the data output. Let $fa.hit \wedge singleton?(fa.hv)$ hold. We set $i := the(fa.hv)$. In addition, we know $cdwb^t \neq 0^B \implies fa.way\_reg = fa.hv$. Since $enc = encf_K(fa.hv)[k-1:0]$ holds by construction of the fully associative cache, the access address of the data RAM of the fully associative cache is given by $radr = encf_K(fa.hv)[k-1:0] \cdot adr[s-1:0]$. In particular, the correctness of the encoder according to lemma 1.4.8 guarantees $radr = \mathrm{bin}_k(i) \cdot adr[s-1:0]$.

Since $sa.hv = fa.hv$ holds, lemma 1.4.4 guarantees for any byte $b \in \mathbb{Z}_B$

$$|sa.dout|_b = |sa.way[i].dout|_b = sa.way[i].data[b][adr[s-1:0]].$$

Additionally, we have

$$|fa.dout|_b = fa.data[b]\,[\mathrm{bin}_k(i) \cdot adr[s-1:0]]$$

and conclude the claim with definition 2.5.1 of equivalence. □

Note that we can claim equality for the data output only under some restrictions. We proceed with a lemma stating that equivalence is preserved in one cycle.

Lemma 2.5.3 *For any set-associative cache sa and any fully associative cache fa, equivalence is preserved if register way_reg is a singleton, i.e., let* sa' *and* fa' *be the next configurations of sa and fa, respectively, given identical input to both configurations. We then have:*

$$fa \equiv sa \wedge singleton?(fa.way\_reg) \implies fa' \equiv sa'$$

Proof: For all the components of the configuration apart from the data RAM, the claims follows immediately from the construction and definition 2.5.1 of equivalence. For the data RAM, the arguments are similar to those of lemma 2.5.2. In the absence of writes, equivalence is trivially preserved; otherwise, we know $radr = encf_K(fa.way\_reg) \cdot adr[s-1:0]$ and, because of the additional assumption, $fa.way\_reg$ is a singleton and equal to $sa.way\_reg$ by the assumption of equivalence. We can then set $i := the(fa.way\_reg)$ and finish the proof just like in lemma 2.5.2. For details, we once again refer to the complete PVS proof. □

Now we know that the relevant outputs of equivalent caches are equal and equivalence is preserved given identical input in the current cycle and a singleton way_reg. This is almost sufficient in order to show consistency of the fully associative cache by induction with the already proved consistency of the set-associative cache. The only problem is the initial cycle where register way_reg may have an arbitrary content and some spurious write signals may be active. As initial configuration for the set-associative cache, we therefore select $sa(fa^1)$ and as input sequence inp' for the set-associative cache, we take the input sequence of the fully associative cache and disable all write signals in the initial cycle except the clear. Note that this trivially leads to $sa[sa(fa^1)]^1 = sa(fa^1)$.

In the following, we use the notation $sa.P$ for a predicate P that holds on the computation the set-associative cache given its input inp'. Similarly, we use $fa.P$ for predicates on the corresponding computation of the fully associative cache.

Proposition 2.5.4 *The following properties hold for any* $t \in \mathbb{N}^+, ad \in \mathbb{B}^a, b \in \mathbb{Z}_B$*:*

$$last_{sa.crd}(t) = last_{fa.crd}(t) \tag{2.20}$$

$$\exists^{last}_{sa.bw(ad,b)}(t) \implies last_{sa.bw(ad,b)}(t) > 0 \wedge \exists^{last}_{fa.bw(ad,b)}(t) \wedge last_{sa.bw(ad,b)}(t) = last_{fa.bw(ad,b)}(t) \tag{2.21}$$

We omit the corresponding proofs due to their triviality. We now show hat associated computations result in equivalent configurations.

Lemma 2.5.5 $\forall t \in \mathbb{N}^+ : fa^t \equiv sa^t$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Since no write signal is active in inp' in cycle 0 by construction, $sa[sa(fa^1)]^1 = sa(fa^1) \equiv fa^1$ trivially holds which concludes the induction base.
Induction step ($\mathbf{t \to t + 1}$): Let equivalence in cycle t hold, i.e, $fa^t \equiv sa^t$. We then have to show equivalence in cycle $t+1$. If $singleton?(fa.way\_reg^t)$ holds, the induction step is concluded with lemma 2.5.3. Let therefore $\neg singleton?(fa.way\_reg^t)$ hold. Since $fa.way\_reg^t = sa.way\_reg^t$ by the equivalence in cycle t, we conclude $clear^{last_{sa.crd}(t)}$ by lemma 2.4.22. Item 3 of $valid\_input?$ than additionally ensures that no write signal is active in cycle t. Thus, equivalence is trivially preserved and the induction step is finished. □

Theorem 2.5.6 *A fully associative cache fulfills the consistency predicate given some input fulfilling* $valid\_input?$.

Proof: By lemma 2.5.5, we conclude $fa^t \equiv sa^t$ for any $t \in N^+$. Lemma 2.5.2 then ensures $fa.hv^t = sa.hv^t$. In particular, this leads to $sa.valid\_input?$ by the definition of inp'. In case of $sa.hit^t$, lemma 2.4.20 additionally guarantees $singleton?(sa.hv^t)$. In order to show the equality of the data output in case of a hit, we need to ensure

$$cdwb \neq 0^B \implies fa.way\_reg = fa.hv$$

If $cdwb^t \neq 0^B$ holds, we know $adr^t =_s adr^{last_{crd}(t)}$ by item 3 of $valid\_input?$. Lemma 2.4.23 for the set-associative cache with $ad := adr^t$ then ensures $sa.way\_reg^{last_{crd}(t)+1} = sa.hv^t$ and with lemma 2.4.2, we additionally conclude $sa.hv^t = sa.way\_reg^t$. By equivalence in cycle t, this leads to $fa.hv^t = fa.way\_reg^t$ and hence, lemma 2.5.2 also guarantees $sa.dout^t = fa.dout^t$.

Finally, data consistency of the set-associative cache according to theorem 2.4.27 and equation 2.5.4 finish the proof. □

2.6 Related work

While there are many published articles on the formal verification of cache consistency protocols, the verification of parameterized cache implementations with different associativity is rarely reported. In the protocol verification [PD96, McM01, SAR99, SSA01] that we will discuss in the related work of the following chapter, caches are abstract units that can hold addresses; each address can at least be valid or dirty which corresponds to our notions in this chapter. From the protocol view, a cache may nondeterministically evict addresses from its memory which coincides with our idea of evicting cache lines. However, caches are simply assumed to be consistent in the sense of definition 1.5.3 if actual data is considered; for the different states, it is assumed that cache addresses stay in the current state unless they are evicted or updated by an explicit request. This basically reflects item 1 and 4 of definition 2.2.1 of extended cache consistency, i.e., that the *hit* and *dirty* signals in our caches are in some sense consistent. In other words, it informally seems that you could use caches as verified in this chapter with these protocol verifications. The fact that the cache consistency properties for an implementation of a sectored K-way set-associative cache in particular are non-trivial seems to be of minor interest to the authors since they do not consider actual implementations.

Chapter 3

A cache memory interface

In this chapter, we present an implementation of a cache memory interface with a read port for instruction fetch and a read/ write port for memory access. We will then formally verify that this implementation is correct according to definition 1.5.2. We start with a formalization of the underlying bus protocol before introducing and finally verifying the actual implementation.

3.1 A bus protocol

We consider the bus protocol presented in [MP00] supporting bursts of length 2^s and single word accesses with words consisting of B bytes. For the VAMP implementation, we have instantiated this bus protocol with $s = 2$ and $B = 8$, i.e., we support burst of four 64-bit words.

The main handshake signals of the bus protocol are *req*, *reqp*, and *brdy*. The CPU raises *req* in order to indicate the beginning of a new request. The memory signals a pending request by asserting *reqp*. Ready data is signalled by the memory *one cycle in advance* by raising *brdy*. Thus, the data supported by the CPU one cycle after *brdy* is written to the main memory in case of a write access. On the other hand, the memory guarantees correct data one cycle after *brdy* in case of a read access. The bus protocol supports the self-explanatory signals *burst* and *wr* as well as well as byte write signals $mwb[B-1:0]$. Typical timing diagrams for burst read- and write accesses are depicted in figures 3.1 and 3.2.

3.1.1 Formal specification

In order to formally verify our cache memory interface, we first have to formalize the bus protocol introduced above. Apart from burst accesses, this formalization is straightforward. We start by introducing some central properties of the bus protocol.

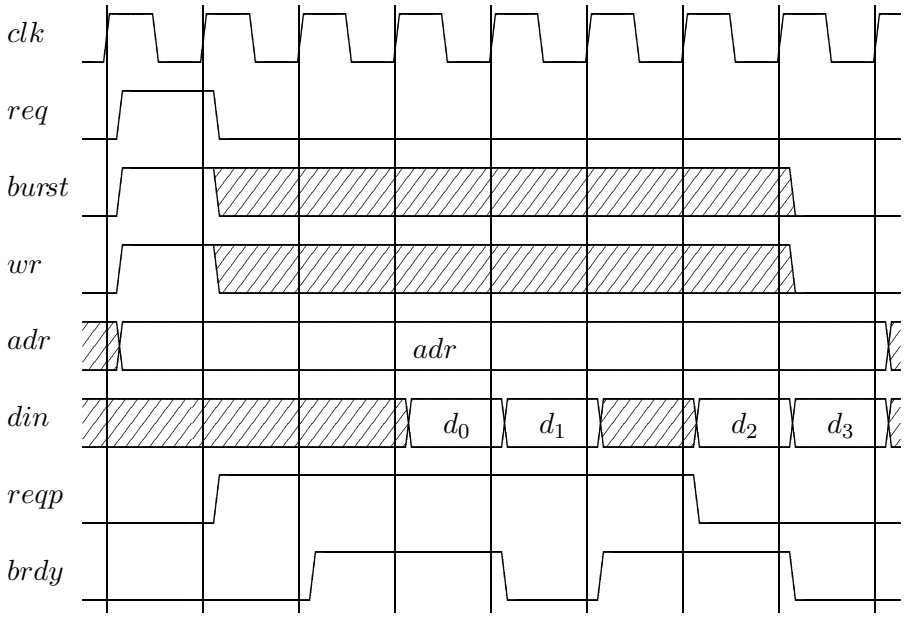

Figure 3.1: 4-burst write timing diagram

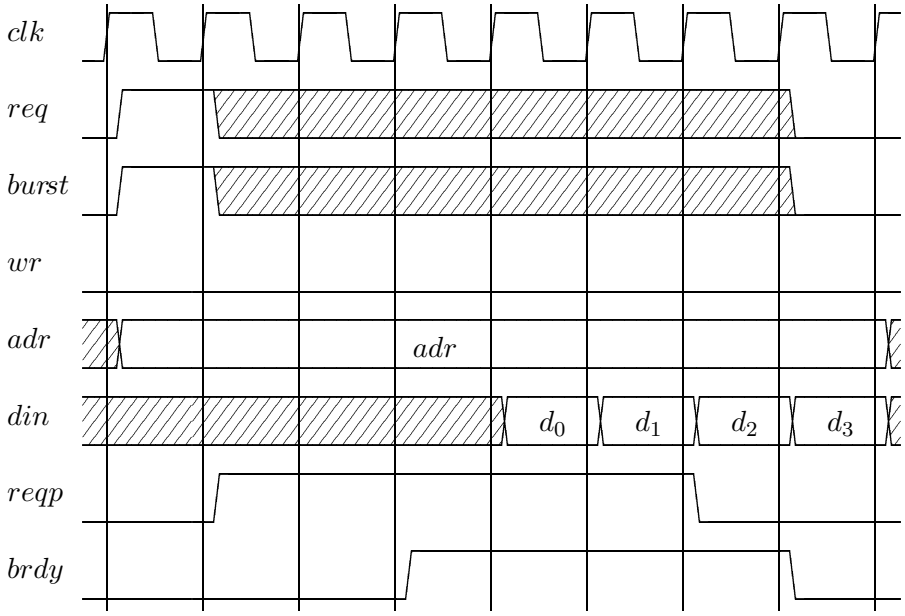

Figure 3.2: 4-burst read timing diagram

First of all, we introduce the notion of a *busy* bus, i.e., the bus is busy in cycle t iff $brdy$ or $reqp$ holds, i.e.,

$$busy^t := brdy^t \vee reqp^t \tag{3.1}$$

We assume that in the initial cycle after power up, the bus is not busy, i.e., $\neg busy^1$. Since requests of a busy bus are ignored, we define a real request in cycle t by

$$request^t := req^t \wedge \neg busy^t \tag{3.2}$$

This allows us to introduce the central invariant for the behaviour of the bus protocol. The bus is busy in cycle $t+1$ iff it was requested in cycle t or an access was pending in cycle t, i.e.,

$$request^t \vee reqp^t \iff busy^{t+1} \tag{3.3}$$

In order to formalize any read access in cycle t, we first need a specification for the memory content in this cycle. The memory content is only updated on writes. The update on single writes is easy since the write address is simply adr^t. However, for bursts, we need some internally generated burst address $badr$ that, informally, counts the number of data words accessed in the burst so far. Formalizing this property in a manner suited to the formal verification turns out to be non-trivial.

Let $badr^t \in \mathbb{B}^a$ be the address of a memory access in cycle t including 'counted' addresses during a burst. For the moment, we leave this address undefined. In order to define the memory content in cycle t, we first introduce a parameterized predicate $mem.bw(adr, b)$ that captures writes to byte b in address adr in analogy to the cache predicate $\$.bw(ad, b)$ introduced in definition 1.5.3 on page 23. Note that since ready data is signalled one cycle in advance and the bus is not busy initially, no byte write can actually occur before cycle 2.

$$\begin{aligned} mem.bw(ad,b)^t \quad := \quad & badr^t = ad \wedge mwb^{t-1}[b] \wedge \\ & t \geq 2 \wedge brdy^{t-1} \wedge mw^{last_{req}(t-1)} \end{aligned} \tag{3.4}$$

The above equation is well founded since we can conclude by induction from $brdy^t$, equations (3.1) to (3.3) and $\neg busy^1$ that $\exists_{req}^{last}(t)$ holds. Now, we can easily specify the memory content in a cycle $t \in \mathbb{N}^+$, i.e, mem^t:

$$\begin{aligned} mem^1 \quad &:= \quad init\_mem \\ \left|mem^{t+1}[\langle ad \rangle]\right|_b \quad &:= \quad \begin{cases} \left|din^t\right|_b & \text{if } mem.bw(ad,b)^t \\ \left|mem^t[\langle ad \rangle]\right|_b & \text{otherwise} \end{cases} \end{aligned} \tag{3.5}$$

Lemma 3.1.1 *For the memory configuration mem, the following alternative non-recursive characterization holds. Formally, we have* $\forall t \in \mathbb{N} \forall b \in$

$\mathbb{Z}_B \forall ad \in \mathbb{B}^a$:

$$\left|mem^t[\langle ad\rangle]\right|_b = \begin{cases} \left|din^{last_{mem.bw(ad,b)}(t)}[\langle ad\rangle]\right|_b & \textit{if } \exists^{last}_{mem.bw(ad,b)}(t) \\ |init\_mem[\langle ad\rangle]|_b & \textit{otherwise} \end{cases}$$

The proof of this lemma follows immediately from proposition 1.2.10. We introduce an additional shorthand notation for the memory content of the address given by $badr$, i.e.,

$$mem\_out^t = mem^t[badr^t] \tag{3.6}$$

We can now easily give the specification of bus protocol accesses where a partial definition of $badr$ is included. We first introduce a predicate that identifies the last cycle in an access, i.e., we set

$$acc\_end := brdy \wedge \neg reqp \tag{3.7}$$

and thus formalize the bus protocol of a single word access by

$$\begin{array}{l} request^t \wedge \neg burst^t \implies \\ \quad \exists^{next}_{acc\_end}(t) \wedge \forall k \in]t : next_{acc\_end}(t)[: reqp^k \wedge \neg brdy^k \wedge \\ \quad badr^{next_{acc\_end}(t)+1} = adr^t \wedge \\ \quad (\neg mw^t \implies dout^{next_{acc\_end}(t)+1} = mem\_out^{next_{acc\_end}(t)+1}) \end{array} \tag{3.8}$$

Note that the effect of a bus protocol write access is already specified by the definition of mem^t from equation (3.5). The cycles in the burst access where $brdy$ is activated are modeled by a function f such that $f(k)$ is the cycle of the k-th $brdy$ in the burst. In this way, we can define for any $k < 2^s$ the burst address with the help of the function f, i.e., in cycle $f(k)+1$, $badr$ simply equals the k-th address in the access. Formally, we have:

$$\begin{array}{l} request^t \wedge burst^t \implies \\ \quad \exists^{next}_{acc\_end}(t) \wedge \exists f : \mathbb{Z}_{2^s} \to]t : next_{acc\_end}(t)] : \\ \quad f(2^s - 1) = next_{acc\_end}(t) \wedge \forall k \in \mathbb{Z}_{2^s-1} : f(k) < f(k+1) \wedge \\ \quad \forall k \in]t : next_{acc\_end}(t)[: reqp^k \wedge (brdy^k \iff k \in f(\mathbb{Z}_{2^s-1})) \wedge \\ \quad \forall k \in \mathbb{Z}_{2^s} : badr^{f(k)+1} = 2^s \cdot \langle adr^t[a-1:s]\rangle + k \wedge \\ \qquad \left(\neg mw^t \implies dout^{f(k)+1} = mem\_out^{f(k)+1}\right) \end{array} \tag{3.9}$$

With these two assumptions, $badr^t$ is defined in all the cycles where it really matters, i.e., where data is either read or written. Figure 3.3 shows an automaton fragment implementing this bus protocol. In the initial state `req`, the signal req is asserted and state `wait` is entered. In reaction to $brdy \wedge reqp$, the state `mem` is entered, and after $brdy \wedge \neg reqp$, `last` is entered. We use this fragment as a generic burst macro in the automata of the memory interface, i.e., any edges into this macro state are edges to state `req` and any outgoing edges from the burst macro are outgoing edges of state `last` of the fragment.

First of all, we want to show some properties of the burst automaton independent of the bus protocol.

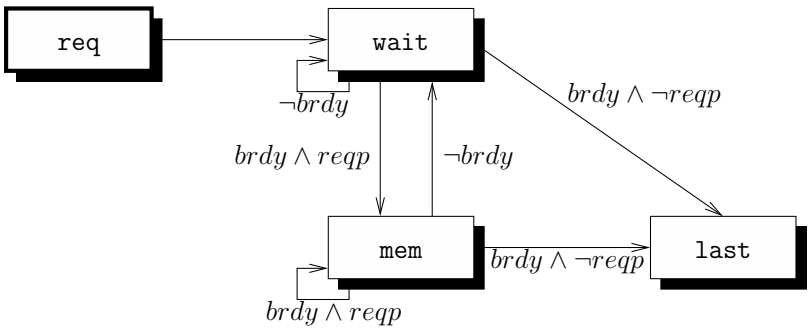

Figure 3.3: Burst control FSD

Lemma 3.1.2 *If the burst automaton is in state* `mem` *or* `wait` *in cycle* t*, there is a cycle* $t' < t$ *where the automation is in state* `req` *and in all intermediate cycles, the automaton is in states* `mem` *or* `wait`*. Formally, we have* $\forall t$*:*

$$\begin{aligned}\mathtt{mem}^t \vee \mathtt{wait}^t \implies \exists t' \in \mathbb{Z}_t : \mathtt{req}^{t'} \wedge \forall t'' \in]t' : t] : \\ \mathtt{wait}^{t''} \wedge (t'' = t' + 1 \vee \neg brdy^{t''-1}) \vee \\ \mathtt{mem}^{t''} \wedge brdy^{t''-1} \wedge reqp^{t''-1}\end{aligned}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 0}$): Since $\mathtt{idle}^0$ holds, there is nothing to show for the induction base.
Induction step ($\mathbf{t \to t + 1}$): Let $\mathtt{mem}^{t+1} \vee \mathtt{wait}^{t+1}$ hold and we have to find some cycle $t' \in \mathbb{Z}_{t+1}$ with

$$\begin{aligned}\mathtt{req}^{t'} \wedge \forall t'' \in]t' : t+1] : \\ \mathtt{wait}^{t''} \wedge (t'' = t' + 1 \vee \neg brdy^{t''-1}) \vee \mathtt{mem}^{t''} \wedge brdy^{t''-1} \wedge reqp^{t''-1}\end{aligned}$$

We split cases on $\mathtt{req}^t$.

1. Let $\mathtt{req}^t$ hold. Cycle $t' := t$ then trivially fulfills the claim because $\mathtt{wait}^{t+1}$ also holds.

2. Let $\neg\mathtt{req}^t$ hold. We then conclude $\mathtt{mem}^t \vee \mathtt{wait}^t$ and thus, we can apply the induction hypothesis to cycle t in order to find a cycle $t' \in \mathbb{Z}_t$ with

   $$\begin{aligned}\mathtt{req}^{t'} \wedge \forall t'' \in]t' : t] : \\ \mathtt{wait}^{t''} \wedge (t'' = t' + 1 \vee \neg brdy^{t''-1}) \vee \mathtt{mem}^{t''} \wedge brdy^{t''-1} \wedge reqp^{t''-1}\end{aligned}$$

   Thus, it only remains to show

   $$\mathtt{wait}^{t+1} \wedge (t + 1 = t' + 1 \vee \neg brdy^t) \vee \mathtt{mem}^{t+1} \wedge brdy^t \wedge reqp^t$$

   Since $t + 1 = t' + 1$ cannot hold in the current case and we already know $\mathtt{mem}^{t+1} \vee \mathtt{wait}^{t+1}$, the claim follows immediately from the possible transitions in the automation fragment according to figure 3.3. □

Corollary 3.1.3 *If the burst automaton is in state* `last` *in cycle* t*, there is a cycle* $t' < t$ *where the automation is in state* `req` *and in all intermediate cycles, the automaton is in states* `mem` *or* `wait`*. Formally, we have* $\forall t$*:*

$$\begin{aligned}&\mathtt{wait}^t \vee \mathtt{mem}^t \vee \mathtt{last}^t \implies \exists t' \in \mathbb{Z}_t : \mathtt{req}^{t'} \wedge \forall t'' \in]t' : t-1] : \\ &\quad \mathtt{wait}^{t''} \wedge (t'' = t'+1 \vee \neg brdy^{t''-1}) \vee \\ &\quad \mathtt{mem}^{t''} \wedge brdy^{t''-1} \wedge reqp^{t''-1}\end{aligned}$$

Proof: In case of $\mathtt{wait}^t \vee \mathtt{mem}^t$, we immediately conclude the proof with lemma 3.1.2. Let therefore $\mathtt{last}^t$ hold. We then conclude $t > 0$ and $\mathtt{mem}^{t-1} \vee \mathtt{wait}^{t-1}$. Thus, we can apply lemma 3.1.2 to cycle $t-1$ in order to conclude the claim. □

Later on, after integrating this automaton fragment into the overall control automata, we will show full compliance with the bus protocol.

3.2 Control automata

In the following, we denote signals and states of the data cache FSD and the data cache itself by the prefix *d*\$ and signals and states of the instruction cache FSD and the instruction cache itself by *i*\$. Note that on an active *clear* input, both automata enter their initial state `idle`. Since we assume valid input from the CPU according to definition 1.5.1, we have an active *clear* only in the initial cycle. Hence, both automata are in state `idle` in cycle 1. We therefore base all our arguments on a starting cycle 1 with initial states in both automata.

We support write-back policy for the data cache and snooping of the other cache in case of a miss, e.g., on an instruction cache miss, the data cache is snooped and, if it holds the corresponding data, invalidated. On the other hand, a hit can be handled locally by a cache without snooping. Therefore, the instruction cache is normally accessed on address *pc* of the memory interface, while the data cache address *adr* is used when the instruction cache is snooped by the data cache. Similarly, the data cache is normally access by *adr*, during a snoop access by address *pc*, and during the write back of a dirty cache line by the eviction address *ev* of the data cache at the beginning of the access which is therefore stored in some register.

The snooping protocol is simple: If one of the caches signals that it wants to snoop the opposite cache by *snoop*, it waits until the opposite cache signals *allow*. As soon as *allow* holds, the snoop operation takes place. In order to avoid deadlocks, we have to ensure that at least one of the caches can signal *allow* if it currently tries to snoop the other cache.

3.2.1 Instruction cache control

Our automaton implementing this snooping for the instruction cache as depicted in figure 3.4 is quite simple; it only needs five states. In the initial

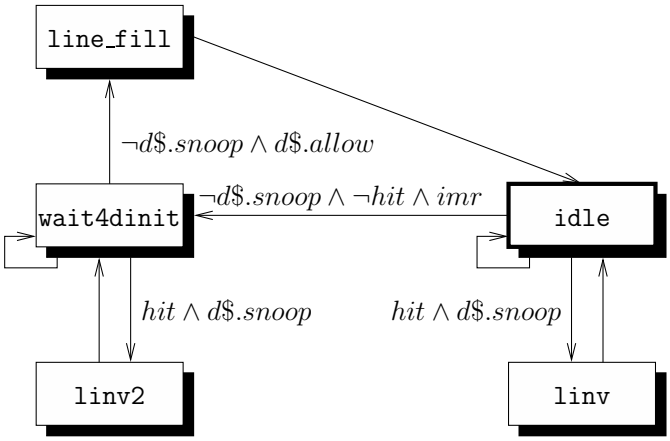

Figure 3.4: Instruction cache control FSD

stage `idle`, we wait for either an instruction port access or a snoop request of the data cache, i.e., $d\$.snoop$. In case of a read hit, the access immediately terminates and we stay in state `idle`. On a snoop hit, a line invalidation is performed in state `linv`. Note that in order to compute whether we actually have a hit in case of a snoop, the instruction cache has to addressed with the data cache address. In case of a miss, state `wait4dinit` is entered and the instruction cache waits for the data cache to be snooped. Note that the instruction cache also can be snooped in this state. This additional snoop state `linv2` is essential in order to avoid deadlocking in case of mutual snoop accesses. After the data cache is snooped, state `line_fill` is entered and a cache line is fetched into the instruction cache. Note that `line_fill` is actually a macro for a burst access as specified in section 3.1, i.e., when `line_fill` is entered, we actually enter sub-state `line_fill.req`, and state `line_fill` is only left from sub-state `line_fill.last`. For the output signals computed by the automaton, the following equations (3.10) hold.

The valid part of the directory is updated during a line invalidation and at the beginning as well as end of a line fill. Only at the end of a line fill is a cache line actually validated and the tag part of the directory written. The instruction cache wants to snoop the data cache in state `wait4dinit` while it allows snooping both in `idle` as well as statewait4dinit. Signal $i\$.snooping$ indicates that the instruction cache is supposed to be addressed with the address from the data cache access port. Signal $i\$.sw$ indicates that the instruction cache is written during a line fill. The beginning of an access, i.e., $i\$.\$rd$, is signalled in states `idle` or `wait4dinit` if a normal access or a snoop access actually starts. Signals $i\$.bcnt\_ce$ and $i.bcnt\_clr$ are the clear and clock enable signals of a counter $i\$.bcnt$ that is used to model the memory address modulo 2^s during a line fill. The cache memory interface only signals $\neg ibusy$ at the end of a line fill or on a read hit in state `idle`.

Since we do not use the instruction cache as write back cache, the dirty input and write signal are tied to 0.

$$
\begin{aligned}
i\$.vw &= \texttt{linv} \vee \texttt{linv2} \vee \texttt{line\_fill.last} \vee \\
&\quad \texttt{line\_fill.req} \\
i\$.val\_in &= \texttt{line\_fill.last} \\
i\$.tw &= \texttt{line\_fill.last} \\
i\$.snoop &= \texttt{wait4dinit} \\
i\$.allow &= \texttt{idle} \vee \texttt{wait4dinit} \\
i\$.snooping &= \texttt{linv} \vee \texttt{linv2} \vee \\
&\quad (\texttt{idle} \vee \texttt{wait4dinit}) \wedge d\$.snoop \\
i\$.sw &= \texttt{line\_fill.mem} \vee \texttt{line\_fill.last} \\
i\$.\$rd &= \texttt{idle} \wedge (d\$.snoop \vee imr) \vee \\
&\quad \texttt{wait4dinit} \wedge (d\$.snoop \vee d\$.allow) \\
i\$.bcnt\_clr &= \texttt{line\_fill.req} \\
i\$.bcnt\_ce &= \texttt{line\_fill.mem} \vee \texttt{line\_fill.last} \\
ibusy &= \neg(\texttt{idle} \wedge \neg d\$.snoop \wedge (i\$.hit \vee \neg imr) \vee \\
&\quad \texttt{line\_fill.last}) \\
i\$.dw &= 0 \\
i\$.dty &= 0 \\
i\$.clear &= clear
\end{aligned}
\tag{3.10}
$$

The remaining inputs of the instruction cache are computed according to equation (3.11). First of all, we have a counter $i\$.bcnt$ which is clocked and cleared by $i\$.bcnt\_ce$ and $i\$.bcnt\_ce$, respectively. The instruction cache is addressed with the input address pc where the s leftmost bits are replaced by $i\$.bcnt$ during a line fill. In case of a snoop, the instruction cache address is given by adr, i.e., the address on the data port; otherwise, we just use pc. The data input of the instruction cache is either given by the memory output or, in case of a dirty snoop hit when the data is just copied from the data cache, $d\$.dout$.

$$
\begin{aligned}
i\$.bcnt' &= \begin{cases} 0^s & \text{if } i\$.bcnt\_clr \\ inc_s(bcnt)[s-1:0] & \text{if } \neg i\$.bcnt\_clr \wedge \\ & \quad i\$.bcnt\_ce \\ i\$.bcnt & \text{otherwise} \end{cases} \\
i\$.adr &= \begin{cases} pc[a-1:s] \cdot i\$.bcnt & \text{if } \texttt{i\$.line\_fill} \\ adr & \text{if } \neg \texttt{i\$.line\_fill} \wedge \\ & \quad i\$.snooping \\ pc & \text{otherwise} \end{cases} \\
i\$.din &= \begin{cases} d\$.dout & \text{if } \texttt{i\$.line\_fill} \wedge \\ & \quad \texttt{d\$.wirte\_back} \\ mem.dout & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.11}
$$

We now show a simple lemmas about this automaton. In chapter 2, we defined a predicate *crd* on cache input that holds iff either of the cache input signals $\$rd$ or *clear* hold. Let therefore $i\$.crd$ hold if *crd* holds for the input of the instruction cache and $d\$.crd$ in analogy for the data cache.

Lemma 3.2.1 *Instruction cache accesses are indeed initiated in states* `idle` *and* `wait4dinit`. *Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned}
&\neg\texttt{idle}^t \wedge \neg\texttt{wait4dinit}^t \implies \\
&\quad last_{i\$.crd}(t) > 0 \wedge \neg clear^{last_{i\$.crd}(t)} \wedge \\
&\quad \texttt{linv}^t ? \, \texttt{idle}^{last_{i\$.crd}(t)} : \texttt{wait4dinit}^{last_{i\$.crd}(t)} \wedge \\
&\quad \forall t' \in]last_{i\$.crd}(t) : t[: \neg\texttt{idle}^{t'} \wedge \neg\texttt{wait4dinit}^{t'} \wedge \\
&\quad \texttt{linv}^t \vee \texttt{linv2}^t \implies last_{i\$.crd}(t) = t - 1 \wedge d\$.snoop^{t-1}
\end{aligned}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): In cycle 1, the automaton is in state `idle` which trivially concludes the induction base.
Induction step ($\mathbf{t \to t + 1}$): Let $\neg\texttt{idle}^{t+1} \wedge \neg\texttt{wait4dinit}^{t+1}$ hold. Thus, we can conclude $\neg clear^t$ and split cases on $\neg\texttt{idle}^t \wedge \neg\texttt{wait4dinit}^t$.

1. Let $\neg\texttt{idle}^t \wedge \neg\texttt{wait4dinit}^t$ hold. This leads to $last_{crd}(t+1) = last_{crd}(t)$. Thus, we can apply the induction hypothesis to cycle t in order to conclude the first four parts of the induction claim. Finally, $\texttt{linv}^{t+1} \vee \texttt{linv2}^{t+1}$ cannot hold because $\neg\texttt{idle}^t \wedge \neg\texttt{wait4dinit}^t$ holds which concludes the proof for this case.

2. Let $\texttt{idle}^t \vee \texttt{wait4dinit}^t$ hold. Hence, we conclude $last_{i\$.crd}(t+1) = t$. Additionally, because of $\texttt{idle}^1$, we conclude $t + 1 > 1$ and thus, $last_{i\$.crd}(t+1) > 0$ which is the first part of the induction claim. We additionally conclude $\texttt{idle}^t$ if $\texttt{linv}^{t+1}$ holds and otherwise, $\texttt{wait4dinit}^t$ with figure 3.4. Finally, $d\$.snoop^t$ follows from $\texttt{linv}^{t+1} \vee \texttt{linv2}^{t+1}$ which concludes the claim since $last_{i\$.crd}(t+1) = t$ holds anyway. □

3.2.2 Data cache control

Since the data cache features write back policy, the corresponding automaton in figure 3.5 is more complex. The initial state is once again called `idle`. A read hit can be handled without leaving this state while on a write hit, state `write` is entered in order to perform the actual write operation in the data cache. A clean miss results in a snoop on the instruction cache in state `wait4snoop` followed by `line_fill` and, potentially, also `write`. On a dirty miss, we have to wait for the memory to become free in state `wait4mem` before entering `write_back` where we access the cache with the eviction address *ev* and write back the dirty cache line to the physical memory. Afterwards, we proceed like in case of a clean miss and therefore, we do not need to

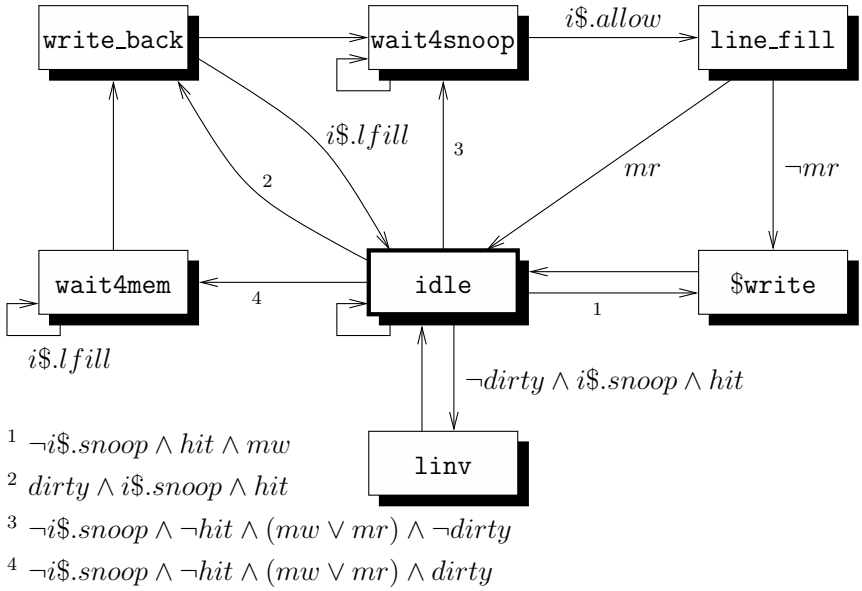

Figure 3.5: Data cache control FSD

invalidate the cache line during a write back since it is invalidated during the following line fill anyway.

On a snoop hit of the instruction cache, there are *two* possibilities. Either it was a clean hit and we just enter `linv` like for the instruction cache automaton or it was a dirty hit and we enter `write_back` since we have to write back the dirty cache line before actually invalidating it.

Note that after such a writeback access in reaction to a snoop, we immediately return to state `idle` and invalidate the corresponding cache line. In order to increase performance for snoop accesses, the instruction cache actually copies the data that the data cache writes back to the memory. Thus, a second memory access is avoided and hence, a line fill in the instruction cache either reads data from the main memory or the data cache.

Note that both `line_fill` and `write_back` are actually super-states consisting of the macro depicted in figure 3.3. For the outputs of the automaton, the following equations (3.12) hold.

The valid part of the data cache is written during a line invalidation and at the beginning or end of a line fill like in the instruction cache. In addition, it is also written at the end of a write back operation if `i$.line_fill` holds, i.e., the instruction cache simultaneously performs a line fill and copies the data from the data cache in case of a dirty snoop hit. Just like in the instruction cache, a line is validated and the tag part of the directory is written only at the end of a line fill. The dirty flag is set in state `$write` and cleared at the end of a line fill. The data cache signals $d\$.snoop$ in state `wait4snoop` while it signals $d\$.allow$ only in the `idle` state. In analogy

to the instruction cache, $d\$.sw$ signals that the data cache is being written during a line fill. A cache access is initiated by $d\$.\$rd$ if we are in state `idle` and either a snoop access or a normal access start. Signal *snooping* is active if the data cache is supposed to be addressed with the instruction cache address, i.e., in the line invalidation state or in the initial cycle of a snoop access. Since we have only one burst counter for both write back and line fill in the data cache, the corresponding signals $d\$.bcnt\_clr$ and $d\$.bcnt\_ce$ have to take both these cases into account. Finally, the data cache automaton signals $\neg dbusy$ on a read hit in state idle, a read at the end of a line fill, and in state `$write`.

$$
\begin{aligned}
d\$.vw &= \texttt{linv} \vee \texttt{line\_fill.last} \vee \texttt{line\_fill.req} \vee \\
&\quad \texttt{write\_back.last} \wedge \texttt{i\$.line\_fill} \\
d\$.val\_in &= \texttt{line\_fill.last} \\
d\$.tw &= \texttt{line\_fill.last} \\
d\$.dty &= \texttt{\$write} \\
d\$.dw &= \texttt{\$write} \vee \texttt{line\_fill.last} \\
d\$.snoop &= \texttt{wait4snoop} \\
d\$.allow &= \texttt{idle} \\
d\$.sw &= \texttt{line\_fill.mem} \vee \texttt{line\_fill.last} \\
d\$.\$rd &= \texttt{idle} \wedge (i\$.snoop \vee mr \vee mw) \\
d\$.snooping &= \texttt{idle} \wedge i\$.snoop \vee \texttt{linv} \\
d\$.bcnt\_clr &= \texttt{write\_back.req} \vee \texttt{line\_fill.req} \\
d\$.bcnt\_ce &= \texttt{write\_back.mem} \vee \texttt{write\_back.last} \vee \\
&\quad \texttt{line\_fill.mem} \vee \texttt{line\_fill.last} \\
dbusy &= \neg(\texttt{idle} \wedge \neg i\$.snoop \wedge \\
&\quad (d\$.hit \wedge mr \vee \neg mr \wedge \neg mw) \vee \\
&\quad \texttt{line\_fill.last} \wedge mr \vee \texttt{\$write}) \\
d\$.clear &= clear
\end{aligned}
\tag{3.12}
$$

The remaining inputs of the data cache are given by the following equation (3.13). The burst counter $d\$.bcnt$ is just the twin of $i\$.bcnt$. The data cache automaton stores the eviction address ev in a register ev_adr at the beginning of a non-snoop access; in case of a snoop, the instruction port address pc is stored. The data cache is addressed with this address ev_adr during a write back, and with adr during a line fill. Note that in both these cases, the rightmost s bits of the address are given by $d\$.bcnt$. In case the data cache is snooped, the data cache is addressed with pc; otherwise, adr is simply used. The data input of the data cache is the data input of the memory interface in state `$write`; in all other states, the output of the memory is selected.

$$
\begin{array}{rcl}
d\$.bcnt' & = & \begin{cases} 0^s & \text{if } d\$.bcnt\_clr \\ inc_s(bcnt)[s-1:0] & \text{if } \neg d\$.bcnt\_clr \wedge \\ & \quad d\$.bcnt\_ce \\ d\$.bcnt & \text{otherwise} \end{cases} \\
ev\_adr' & = & \begin{cases} pc & \text{if } d\$.\$rd \wedge d\$.snooping \\ d\$.ev & \text{if } d\$.\$rd \wedge \neg d\$.snooping \\ ev\_adr & \text{otherwise} \end{cases} \\
d\$.adr & = & \begin{cases} adr[a-1:s] \cdot d\$.bcnt & \text{if } \texttt{d\$.line\_fill} \\ ev\_adr[a-1:s] \cdot d\$.bcnt & \text{if } \texttt{d\$.write\_back} \\ pc & \text{if } d\$.snooping \\ adr & \text{otherwise} \end{cases} \\
d\$.din & = & \texttt{d\$.\$write}?\, din : mem.dout
\end{array}
\tag{3.13}
$$

We now show a basic lemma about the data cache automaton.

Lemma 3.2.2 *Data cache accesses are indeed initiated in state* `idle`*, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$
\neg \texttt{idle}^t \implies last_{d\$.crd}(t) > 0 \wedge \neg clear^{last_{d\$.crd}(t)} \wedge \forall t' \in]last_{d\$.crd}(t) : t[: \neg \texttt{idle}^{t'}
$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Initially, the automaton is in state `idle` which trivially concludes the induction base.
Induction step ($\mathbf{t \to t + 1}$): Let $\neg \texttt{idle}^{t+1}$ hold. We trivially conclude $\neg clear^t$ and split cases on $\texttt{idle}^t$.

1. Let $\texttt{idle}^t$ hold. From $\neg \texttt{idle}^{t+1}$, figure 3.5, and equation (3.12), we conclude $\$rd^t$ and thus, $last_{d\$.crd}(t+1) = t$. Since $\texttt{idle}^1$ holds, we also conclude $t + 1 > 1$, i.e., $last_{d\$.crd}(t+1) > 0$. Thus, this case of the claim is already finished since $\neg clear^t$ holds.

2. Let $\neg \texttt{idle}^t$ hold. From $\neg \texttt{idle}^{t+1}$, we can also conclude $\neg clear^t$. Thus, we can apply the induction hypothesis to cycle t in order to conclude this case of the claim. □

3.3 Data paths

Figure 3.6 shows the top-level data paths of the cache memory interface. Note that both circuits *adr_gen* contain some registers *bcnt* in order to emulate the computation of the burst address *badr* from the bus protocol.

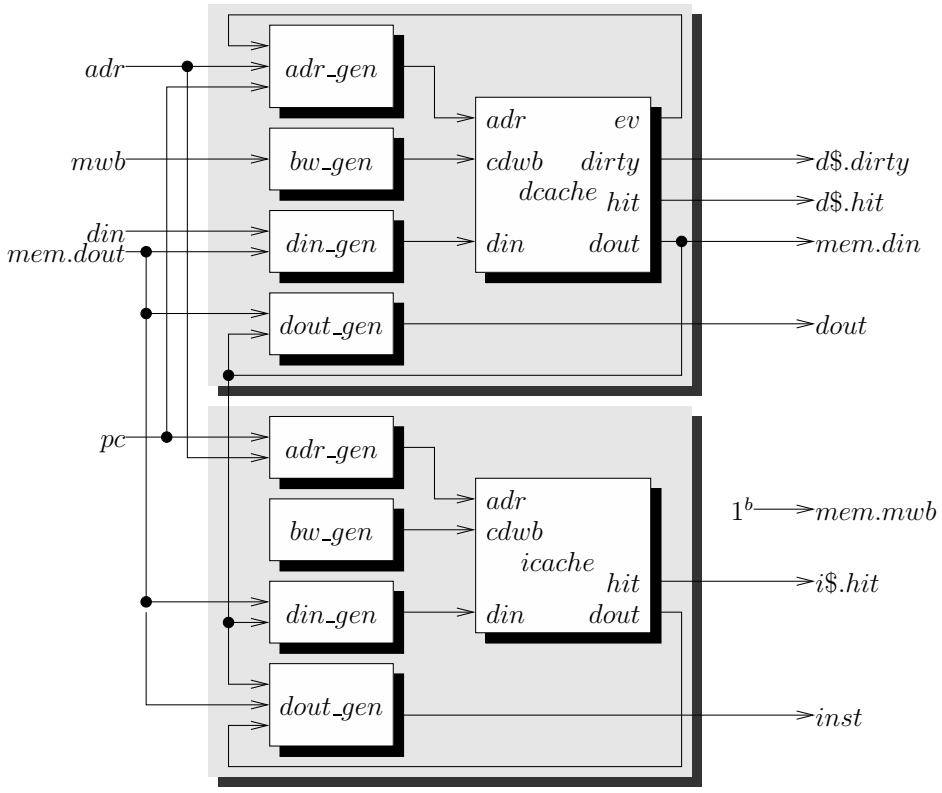

Figure 3.6: Top-level data paths of the cache memory interface

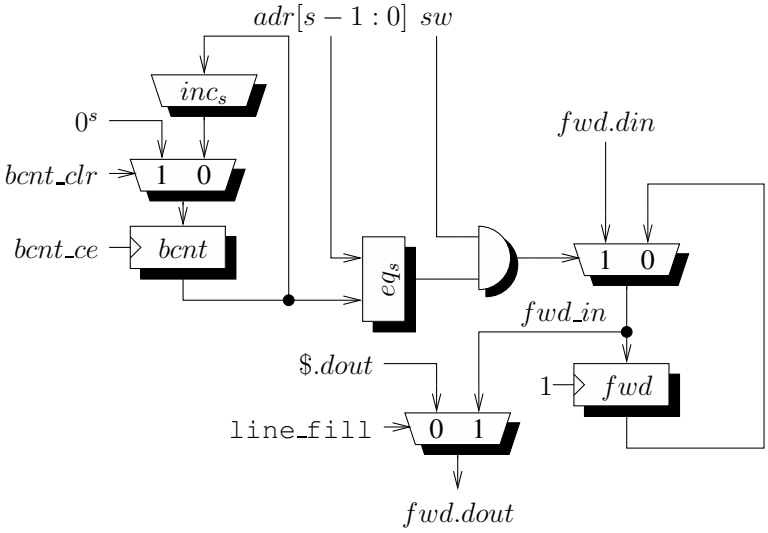

Figure 3.7: Forwarding circuit of the cache memory interface

Additionally, the eviction address ev has to be stored inside some register in $d\$.adr\_gen$ in order to allow for write back of dirty cache lines. Note that we support forwarding for read accesses from the memory; therefore, both circuits $dout\_gen$ contain a register that can store a 64-bit data word. Finally, there is also a register inside $d\$.gen\_bw$ that stores the byte write signals in the current access. By putting the two automata from the last section together, we obtain the complete automaton for the cache memory interface. Figure 3.7 depicts the general sub-circuit used for forwarding data in both caches; equations (3.14) and (3.15) actually give the corresponding equations for both caches. In addition, a register $cdwb$ stores the memory byte write signals at the beginning of an access.

$$\begin{array}{rcl} i\$.fwd\_in & = & i\$.sw \wedge eq_s(i\$.bcnt, adr[s-1:0])? \\ & & i\$.din : i\$.fwd \\ i\$.fwd' & = & i\$.fwd\_in \\ adr\_d & = & \texttt{d\$.wirte\_back}?\, ev\_adr : adr \\ d\$.fwd\_in & = & d\$.sw \wedge eq_s(d\$.bcnt, adr\_d[s-1:0])? \\ & & mem.dout : d\$.fwd \\ d\$.fwd' & = & d\$.fwd\_in \\ cdwb' & = & d\$.\$rd?\, mwb : cdwb \end{array} \tag{3.14}$$

For the output to the CPU, we select either fwd_in as defined above or $dout$ of the corresponding cache depending on whether a line fill is currently performed.

$$\begin{array}{rcl} dout & = & \texttt{d\$.line\_fill}?\, d\$.fwd\_in : d\$.dout \\ inst & = & \texttt{i\$.line\_fill}?\, i\$.fwd\_in : i\$.dout \end{array} \tag{3.15}$$

Finally, the memory signals are summarized in equation (3.16). The memory is requested in the `req` state of the three burst macros; the memory write signals is only active during write back. The data input of the memory is always given by the data output of the data cache. During a write back, the memory is addressed on the eviction address ev_adr, during a line fill in the instruction cache by pc; otherwise, we chose just adr.

$$\begin{array}{lcl} mem.req & = & \texttt{d\$.line\_fill.req} \vee \texttt{d\$.write\_back.req} \vee \\ & & \quad \texttt{i\$.line\_fill.req} \\ mem.burst & = & 1 \\ mem.wr & = & \texttt{d\$.write\_back} \\ mem.mwb & = & 1^B \\ mem.din & = & d\$.dout \\ mem.adr & = & \begin{cases} ev\_adr & \text{if } \texttt{d\$.wirte\_back} \\ pc & \text{if } \neg\texttt{d\$.wirte\_back} \wedge \\ & \quad \texttt{i\$.line\_fill} \\ adr & \text{otherwise} \end{cases} \end{array} \tag{3.16}$$

3.4 Correctness proof

The main proof goal in this section is given by definition 1.5.2 of a correct memory interface. Not in particular that as a precondition for a correct memory interface, we have valid input from the CPU according to definition 1.5.1 and a bus protocol as defined in section 3.1.1. We will use both these assumptions in all the proofs of this section without adding them as explicit assumptions to the left-hand side of each implication.

In order to actually prove the cache memory interface correct, we first verify that the automata fit the bus protocol we specified and that for both instruction and data cache, the cache memory interface produces valid input according to definition 2.2.2 such that we can actually use cache consistency for both these caches. We will then show a central consistency invariant and derive the correctness of the memory interface from it.

3.4.1 Valid cache input and bus protocol compliance

We start with a couple of lemmas about the possible combinations of states in the two automata.

Lemma 3.4.1 *Instruction and data cache cannot perform a line fill operation simultaneously, i.e., we have formally:*

$$\forall t \in \mathbb{N}^+ : \neg(\texttt{d\$.line\_fill}^t \wedge \texttt{i\$.line\_fill}^t)$$

Proof: Induction base ($\mathbf{t = 1}$): In the initial cycle, both automata are in their respective idle states which concludes the induction base.
Induction step ($\mathbf{t \to t + 1}$): Let $\neg(\texttt{d\$.line\_fill}^t \wedge \texttt{i\$.line\_fill}^t)$ hold. We conclude $\neg(\texttt{d\$.line\_fill}^{t+1} \wedge \texttt{i\$.line\_fill}^{t+1})$ by contradiction, i.e., we assume that $\texttt{d\$.line\_fill}^{t+1} \wedge \texttt{i\$.line\_fill}^{t+1}$ holds. We split cases on $\texttt{i\$.line\_fill}^t$.

1. Let $\neg\texttt{i\$.line\_fill}^t$ hold. Since $\texttt{i\$.line\_fill}^{t+1}$ holds, we conclude from figure 3.4 that $\neg d\$.snoop^t \wedge d\$.allow^t$ holds which equals $\texttt{d\$.idle}^t$ according to equations (3.12). According to figure 3.5, this is a contradiction to $\texttt{d\$.line\_fill}^{t+1}$ and thus finishes this case of the claim.

2. Let $\texttt{i\$.line\_fill}^t$ hold. Because of the induction hypothesis, this leads to $\neg\texttt{d\$.line\_fill}^t$. Since $\texttt{d\$.line\_fill}^{t+1}$ holds, we conclude from figure 3.5 that $i\$.allow^t$ holds which is equivalent to $\texttt{i\$.idle}^t \vee \texttt{i\$.wait4dinit}^t$ because of equations (3.10) which is a contradiction to $\texttt{i\$.line\_fill}^t$. □

Lemma 3.4.2 *A line write access in the data cache and a line fill access in the instruction cache can only occur "synchronously" after a dirty snoop hit, i.e., it holds that*

$$\begin{aligned}
&\forall t \in \mathbb{N}^+ : \texttt{i\$.line_fill}^t \wedge \texttt{d\$.write_back}^t \implies \\
&\quad (\texttt{i\$.line_fill.req}^t \wedge \texttt{d\$.write_back.req}^t \vee \\
&\quad \texttt{i\$.line_fill.wait}^t \wedge \texttt{d\$.write_back.wait}^t \vee \\
&\quad \texttt{i\$.line_fill.mem}^t \wedge \texttt{d\$.write_back.mem}^t \vee \\
&\quad \texttt{i\$.line_fill.last}^t \wedge \texttt{d\$.write_back.last}^t) \wedge \\
&\quad last_{i\$.crd}(t) = last_{d\$.crd}(t) \wedge \texttt{i\$.wait4dinit}^{last_{i\$.crd}(t)} \wedge \\
&\quad d\$.hit^{last_{i\$.crd}(t)} \wedge d\$.dirty^{last_{i\$.crd}(t)}
\end{aligned}$$

Proof: The claim is shown by induction on t.
Induction base ($\mathbf{t = 1}$): Since both automata are initially in the idle state, the induction base holds trivially.
Induction step ($\mathbf{t \to t + 1}$): We have to show that on $\texttt{i\$.line\_fill}^{t+1} \wedge \texttt{d\$.write\_back}^{t+1}$, both automata are in the same sub-state in cycle $t + 1$ and the claim about $last_{i\$.crd}(t + 1)$ hold. We split cases on whether we can apply the induction hypothesis to cycle t.

1. Let $\texttt{i\$.line\_fill}^t \wedge \texttt{d\$.write\_back}^t$ hold. We trivially conclude $last_{i\$.crd}(t + 1) = last_{i\$.crd}(t)$ and $last_{d\$.crd}(t + 1) = last_{d\$.crd}(t)$. The induction hypothesis then guarantees that both automata are in the same sub-state in cycle t and the claims about $last_{i\$.crd}(t + 1)$ also follows from the induction hypothesis. Since states `i$.line_fill` and `d$.write_back` both consist of the burst macro sharing common input signals form the physical memory, both automata are either in the

same sub-state in cycle $t+1$ or they leave the burst macro and neither $\texttt{i\$.line\_fill}^{t+1}$ nor $\texttt{d\$.write\_back}^{t+1}$ holds which concludes the claim.

2. Let $\neg(\texttt{i\$.line\_fill}^t \wedge \texttt{d\$.write\_back}^t)$ hold. We split cases on $\texttt{i\$.line\_fill}^t$.

   (a) Let $\neg\texttt{i\$.line\_fill}^t$ hold. Since $\texttt{i\$.line\_fill}^{t+1}$ holds, we conclude from figure 3.4 that $\texttt{i\$.line\_fill.req}^{t+1}$ and $\neg d\$.snoop^t \wedge d\$.allow^t$ hold which equals $\texttt{d\$.idle}^t$ according to equations (3.12). This leads to $last_{d\$.crd}(t+1) = t$. Since $\texttt{d\$.write\_back}^{t+1}$ holds, we conclude with figure 3.5 that $i\$.snoop^t \wedge d\$.hit^t \wedge d\$.dirty^t$ holds which leads to $\texttt{i\$.wait4dinit}^t$, i.e., $last_{i\$.crd}(t+1) = t$. Additionally, it follows that $\texttt{d\$.write\_back.req}^{t+1}$ holds. Because of $\texttt{i\$.line\_fill}^{t+1}$, this case is proved.

   (b) Let $\texttt{i\$.line\_fill}^t$ hold. This leads to $\neg\texttt{d\$.write\_back}^t$. Because of $\texttt{d\$.write\_back}^{t+1}$, we conclude from figure 3.5 that $snoop\_i^t$ holds which is equal to $\texttt{i\$.wait4dinit}^t$. This is a contradiction and thus finishes the claim. □

Lemma 3.4.3 *The memory is busy if and only if at least one of the two automata is in a busy state, i.e.,*

$$\begin{aligned}\forall t \in \mathbb{N}^+ : busy^t \iff\ & (\texttt{i\$.line\_fill.wait}^t \vee \texttt{i\$.line\_fill.mem}^t \vee \\ & \texttt{d\$.line\_fill.wait}^t \vee \texttt{d\$.line\_fill.mem}^t \vee \\ & \texttt{d\$.write\_back.wait}^t \vee \texttt{d\$.write\_back.mem}^t)\end{aligned}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Initially, the memory is *not* busy according to the bus protocol and both automata are in the idle state. This concludes the induction base.
Induction step ($\mathbf{t \to t+1}$): We have to show

$$\begin{aligned}busy^{t+1} \iff\ & (\texttt{i\$.line\_fill.wait}^{t+1} \vee \texttt{i\$.line\_fill.mem}^{t+1} \vee \\ & \texttt{d\$.line\_fill.wait}^{t+1} \vee \texttt{d\$.line\_fill.mem}^{t+1} \vee \\ & \texttt{d\$.write\_back.wait}^{t+1} \vee \texttt{d\$.write\_back.mem}^{t+1})\end{aligned}$$

and we split cases on $busy^t$.

1. Let $busy^t$ hold. This leads to $\neg request^t$ because of equation (3.2) and thus, we have $busy^{t+1} \iff reqp^t$ according to equation (3.3). With the induction hypothesis, we conclude that we are in one of the six busy states in cycle t. However, it could still be the case that we are in one of the request states, but the request is ignored because of $busy^t$. Since we are in a busy state in cycle t, we can only be in a request

state in the same cycle if one automaton is in a busy state while the other is in a request state. If $\texttt{i\$.line\_fill.req}^t$ holds, lemma 3.4.1 guarantees $\neg\texttt{d\$.line\_fill}^t$ which leads to $\texttt{d\$.write\_back}^t$ because we are in a busy state. However, in this case, lemma 3.4.2, guarantees $\texttt{d\$.write\_back.req}^t$ which is a contradiction since it is not a busy state. Hence, $\texttt{i\$.line\_fill.req}$ cannot hold.

By arguing along the same lines, we can also conclude that the data cache is not in a request state in cycle t, i.e.,

$$\neg\texttt{d\$.line\_fill.req}^t \wedge \neg\texttt{d\$.write\_back.req}^t$$

In other words, we are in some memory state, but not in an initial one in cycle t. Thus, we remain in a busy state in cycle $t+1$ if and only if $reqp^t$ holds which concludes this case of the claim.

2. Let $\neg busy^t$ hold. With the induction hypothesis, we conclude that we are in none of the six busy states in cycle t. According to equation (3.3), $busy^{t+1}$ holds iff $request^t$ holds which equals req^t because of $\neg busy^t$. From equation (3.16), we then conclude $req^t = i\$.req^t \vee d\$.req^t$ which leads to

$$busy^{t+1} \iff \texttt{i\$.line\_fill.req}^t \vee \texttt{d\$.line\_fill.req}^t \vee \texttt{d\$.write\_back.req}^t$$

According to figure 3.3, the state after one of the above three request states is on of the six busy states which concludes the claim. □

Lemma 3.4.4 *In the cycle after the end of a memory access, the automaton is in one of the* `last` *states, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$(brdy^t \wedge \neg reqp^t) \iff (\texttt{d\$.line\_fill.last}^{t+1} \vee \texttt{d\$.write\_back.last}^{t+1} \vee \texttt{i\$.line\_fill.last}^{t+1})$$

Proof: The claim is proved by induction on t. Since it uses the same arguments as the proof of the previous lemma, we omit further details in this thesis. □

Now, we know that no memory request is lost because there is no request of a busy memory. This is an important step in proving the correctness of the bus protocol implementation.

As a next step, we want to show that the input generated for data- and instruction cache fulfills predicate *valid_input?* according to definition 2.2.2 on page 29. Note that in order to prove *valid_input?*, we actually need part of the bus protocol correctness in order for a line fill to be complete according to *valid_input?*.

Lemma 3.4.5 *If the instruction cache is neither in state* `i$.idle` *nor in state* `i$.linv`*, the instruction memory read signal is active and the signals at the instruction port equal the values they had at the beginning of the access, i.e.,*

$$\forall t \in \mathbb{N}^+ : \neg \texttt{i\$.idle}^t \wedge \neg \texttt{i\$.linv}^t \implies imr^t \wedge \{pc, imr\}^t = \{pc, imr\}^{last_{i\$.crd}(t)}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Initially, the automaton is in state `i$.idle` which trivially fulfills the claim.
Induction step ($\mathbf{t \to t + 1}$): Let $\neg \texttt{i\$.idle}^{t+1} \wedge \neg \texttt{i\$.linv}^{t+1}$ hold. We then have to show

$$imr^{t+1} \wedge \{pc, imr\}^{t+1} = \{pc, imr\}^{last_{i\$.crd}(t+1)}$$

in order to conclude the claim. We split cases on $\texttt{i\$.idle}^t$.

1. Let $\texttt{i\$.idle}^t$ hold. With figure 3.4, we then conclude $\neg d\$.snoop^t \wedge \neg i\$.hit^t \wedge imr^t$. This trivially leads to $last_{i\$.crd}(t+1) = t$. Because of equation (3.10), we can conclude $ibusy^t$ and thus, because of the input stability from definition 1.5.1 on page 16, we have $\{pc, imr\}^t = \{pc, imr\}^{t+1}$ and this case is proved.

2. Let $\neg \texttt{i\$.idle}^t$ hold. We conclude $\neg \texttt{i\$.linv}^t \wedge \neg \texttt{i\$.line\_fill.last}^t$ from figure 3.4 and thus, the induction premise guarantees

   $$imr^t \wedge \{pc, imr\}^t = \{pc, imr\}^{last_{i\$.crd}(t)}.$$

   Since $ibusy^t$ holds according to equation (3.10), we use definition 1.5.1 in order to conclude $\{pc, imr\}^t = \{pc, imr\}^{t+1}$. Since $last_{i\$.crd}(t+1)$ either equals t or $last_{i\$.crd}(t)$, this concludes the claim. □

Lemma 3.4.6 *The data cache port signals dbusy from the beginning of the access on, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned}&\neg \texttt{d\$.idle}^t \wedge (mw^{last_{d\$.crd}(t)} \vee mr^{last_{d\$.crd}(t)}) \implies \\ &\quad \{adr, din, mw, mr, mwb\}^t = \{adr, din, mw, mr, mwb\}^{last_{d\$.crd}(t)} \wedge \\ &\quad \forall t' \in [last_{d\$.crd}(t) : t[: dbusy^t\end{aligned}$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): Since the automaton is initially in state `d$.idle`, the induction base holds trivially.
Induction base ($\mathbf{t \to t + 1}$): Let $\neg \texttt{d\$.idle}^{t+1}$ and $mw^{last_{d\$.crd}(t+1)} \vee mr^{last_{d\$.crd}(t+1)}$ hold. We split cases on $\texttt{d\$.idle}^t$.

1. Let $\texttt{d\$.idle}^t$ hold. We trivially conclude $last_{d\$.crd}(t+1) = t$. Since we have $mw^t \vee mr^t$, $\texttt{d\$.idle}^t$, and $\neg \texttt{d\$.idle}^{t+1}$, we get $dbusy^t$ with figure 3.5 and equation (3.12). This concludes the claim with definition 1.5.1.

2. Let $\neg$d\$.idle^t hold. We trivially conclude $last_{d\$.crd}(t+1) = last_{d\$.crd}(t)$ and apply the induction hypothesis in order to obtain

$$\{adr, din, mw, mr, mwb\}^t = \{adr, din, mw, mr, mwb\}^{last_{d\$.crd}(t)} \wedge$$
$$\forall t' \in [last_{d\$.crd}(t) : t[: dbusy^t$$

Since $\neg$d\$.idle^{t+1} holds, we conclude $dbusy^t$ with figure 3.5 and equation (3.12). This concludes the claim with the help of definition 1.5.1. □

Corollary 3.4.7 *If the date cache automaton performs a line fill, a cache write, or is in one of the synchronization states* wait4mem *and* wait4snoop*, the memory read or write signal was active at the beginning of the access and the input on the data port has remained stable since then. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\text{d\$.wait4snoop}^t \vee \text{d\$.wait4mem}^t \vee \text{d\$.cache\_write}^t \vee \text{d\$.line\_fill}^t \implies$$
$$(mw^t \vee mr^t) \wedge \neg \text{i\$.wait4dinit}^{last_{i\$.crd}(t)} \wedge$$
$$\{adr, din, mw, mr, mwb\}^t = \{adr, din, mw, mr, mwb\}^{last_{d\$.crd}(t)} \wedge$$
$$\forall t' \in [last_{d\$.crd}(t) : t[: dbusy^{t'}$$

Proof: If $(mw^{last_{d\$.crd}(t)} \vee mr^{last_{d\$.crd}(t)}) \wedge \neg$i\$.wait4dinit$^{last_{i\$.crd}(t)}$ holds, lemma 3.4.6 can be applied and together with definition 1.5.1, the claim is concluded. We therefore set $l := last_{d\$.crd}(t)$ and only have to show that $(mw^l \vee mr^l) \wedge \neg$i\$.wait4dinit^l holds, i.e., we assume $\neg mw^t \wedge \neg mr^t \vee$ i\$.wait4dinit^l and construct a contradiction. We know that d\$.idle^l holds while d\$.idle^t does not hold. According to figure 3.5, the only way to leave state d\$.idle without either active mw or mr signals is by entering states d\$.linv or d\$.write_back.req. However, if d\$.linv^{l+1} holds, we conclude d\$.idle^{l+2} which is a contradiction according to lemma 3.2.2 because $\forall t' \in]last_{i\$.crd}(t) : t[: \neg$d\$.idle$^{t'}$ holds. Let therefore d\$.write_back.req^{l+1} hold. In this case, lemma 3.4.2 ensures that i\$.line_fill.req^{l+1} also holds. Since we have $\neg$d\$.write_back^t, there is some cycle $t' \in]l : t[$ such that d\$.write_back.last$^{t'}$ holds. With lemma 3.4.2, we additionally conclude i\$.line_fill.last$^{t'}$ and thus, d\$.idle$^{t'+1}$ holds which is a contradiction to lemma 3.2.2 since d\$.idle$^{t'+1}$ cannot hold. □

Lemma 3.4.8 *If the instruction cache is in a line invalidation state, the instruction cache address equals the address at the cycle of the beginning of the instruction port access, i.e., we have* $\forall t \in \mathbb{N}^+$*:*

$$\text{i\$.linv}^t \vee \text{i\$.linv2}^t \implies i\$.adr^t = i\$.adr^{last_{i\$.crd}(t)}$$

Proof: Let i\$.linv$^t \vee$ i\$.linv2^t. We trivially conclude $last_{i\$.crd}(t) = t - 1$ and $d\$.snoop^{t-1}$ from figure 3.4. With equations (3.10) and (3.12), we additionally conclude d\$.wait4snoop^{t-1} and d\$.line_fill.req^t. Thus, equations (3.11) and (3.13) ensure $i\$.adr^t = d\$.adr^t$ and $i\$.adr^{t-1} = d\$.adr^{t-1}$.

By applying lemma 3.4.7 for both cycles $t-1$ and t, we conclude $d\$.adr^{t-1} = d\$.adr^t$ and thus, the proof is finished. □

Lemma 3.4.9 *The input of the instruction cache fulfills the first three items of predicate valid_input?, i.e., we have:*

1. $i\$.clear^0$
2. $\forall t \in \mathbb{N}^+ : i\$.vw^t \wedge i\$.val\_in^t \implies i\$.tw^t$
3. $\forall t \in \mathbb{N}^+ : i\$.vw^t \vee i\$.tw^t \vee i\$.dw^t \vee \exists l \in \mathbb{Z}_B : i\$.cdwb^t[l] \implies i\$.adr^t =_s i\$.adr^{last_{i\$.crd}(t)} \wedge \neg i\$.clear^{last_{i\$.crd}(t)} \wedge \neg i\$.\$rd^t$

Proof: The first item, i.e., $i\$.clear^0$ is trivially fulfilled since $i\$.clear = clear$ holds and definition 1.5.1 guarantees $clear^0$. Similarly, the second item can immediately be concluded from equation (3.10). Thus, only the third item requires real consideration. Let $i\$.vw^t$, $i\$.tw^t$, $i\$.dw^t$, or $i\$.cdwb^t[l]$ hold for some $l \in \mathbb{Z}_B$. According to equation (3.10), the instruction cache automaton is either in state $\texttt{i\$.line\_fill}^t$, $\texttt{i\$.linv}^t$, or $\texttt{i\$.linv2}^t$. Hence, we can trivially conclude $\neg i\$.\$rd^t$. We set $l := last_{i\$.crd}(t)$ and with lemma 3.2.1, we additionally get $\neg i\$.clear^l$. Thus, we only have to show $i\$.adr^t =_s i\$.adr^l$.

1. Let $\neg\texttt{i\$.line\_fill}^t$ hold. We can then apply lemma 3.4.8 in order to conclude $adr^t = adr^l$ which finishes this case of the claim
2. Let $\texttt{i\$.line\_fill}^t$ hold. We apply lemma 3.4.5 in order to conclude $pc^t = pc^l$. With lemma 3.2.1, we additionally get $\texttt{i\$.wait4dinit}^l$. If $\texttt{i\$.linv2}^{l+1}$ holds, we trivially conclude $\texttt{i\$.wait4dinit}^{l+2}$ which is a contradiction to lemma 3.2.1. Hence, we know $\texttt{i\$.line\_fill}^{l+1}$ holds and figure 3.4 and equation (3.12) ensure that $d\$.allow^l$ holds and thus, $\texttt{d\$.idle}^l$. However, this leads to $\neg i\$.snooping^l$ according to equation (3.10) and thus, equation (3.11) ensures both $i\$.adr^l = pc^l$ and $i\$.adr^t =_s pc^t$ which finishes the proof. □

Lemma 3.4.10 *The input of the data cache fulfills the first three items of predicate valid_input?, i.e., we have:*

1. $d\$.clear^0$
2. $\forall t \in \mathbb{N}^+ : d\$.vw^t \wedge d\$.val\_in^t \implies d\$.tw^t$
3. $\forall t \in \mathbb{N}^+ : d\$.vw^t \vee d\$.tw^t \vee d\$.dw^t \vee \exists l \in \mathbb{Z}_B : d\$.cdwb^t[l] \implies d\$.adr^t =_s d\$.adr^{last_{d\$.crd}(t)} \wedge \neg d\$.clear^{last_{d\$.crd}(t)} \wedge \neg d\$.\$rd^t$

Proof: The arguments for showing this claim are very similar to those of lemma 3.4.9; they are therefore omitted in this thesis. □

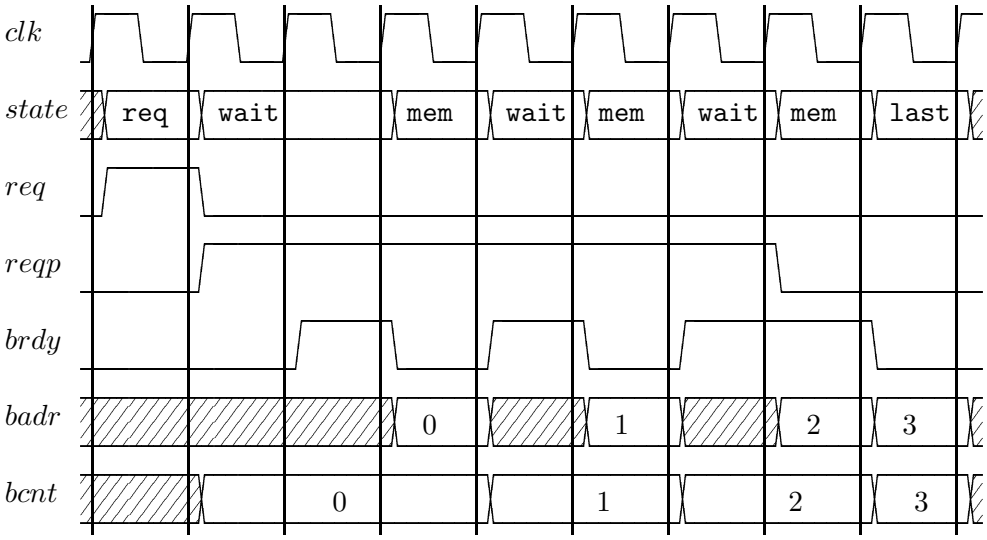

Figure 3.8: Correctness of the burst FSD

In order to show the remaining items of *valid_input?*, we have to take a look at the correctness of a memory access. In particular, we have to show that the addresses generated by the control in order to access the caches in a memory burst access match the addresses specified by the bus protocol. It is then easy to show that any line fill is complete since a burst access is complete according to the bus protocol and it accesses all addresses in a cache line.

Lemma 3.4.11 *During a data cache line fill, the physical memory is accessed in states* `d$.line_fill.mem` *and* `d$.line_fill.last`*; there are exactly* 2^s *such accesses as specified by the bus protocol. Formally, we claim for any cycle* $t \in \mathbb{N}^+$*:*

$$\begin{aligned}
&\texttt{d\$.line_fill.last}^t \implies \exists last_{req}(t) \wedge last_{d\$.crd}(t) < last_{req}(t) \wedge \\
&\quad \texttt{d\$.line_fill.req}^{last_{req}(t)} \wedge \exists g : \mathbb{Z}_{2^s} \to]last_{req}(t) : t] : \\
&\quad g(2^s - 1) = t \wedge \forall i \in \mathbb{Z}_{2^s - 1} : g(i) < g(i+1) \wedge \\
&\quad \forall i \in \mathbb{Z}_{2^s} : badr^{g(i)} = 2^s \cdot \langle adr^{last_{req}(t)}[a-1:s] \rangle + i \wedge \\
&\qquad mem.dout^{g(i)} = mem^{g(i)}[badr^{g(i)}] \wedge \\
&\quad \forall t' \in]last_{req}(t) : t[: t' \in g(\mathbb{Z}_{2^s - 1})? \\
&\qquad \texttt{d\$.line_fill.mem}^{t'} : \texttt{d\$.line_fill.wait}^{t'}
\end{aligned}$$

Proof: Figure 3.8 depicts the timing of a burst access together with automaton states and the value of *badr* mod 2^s and thus illustrates the main part of the lemma's claim. Let $\texttt{d\$.line\_fill.last}^t$ hold. We apply lemma 3.1.3 in

order to find some cycle $t' < t$ with $\texttt{d\$.line\_fill.req}^{t'}$ and $\forall t'' \in]t' : t-1]$, we have

$$\texttt{wait}^{t''} \wedge (t'' = t' + 1 \vee \neg brdy^{t''-1}) \vee \texttt{mem}^{t''} \wedge brdy^{t''-1} \wedge reqp^{t''-1}$$

Hence, we have $req^{t'}$ and lemma 3.4.3 guarantees $\neg busy^{t'}$. Additionally, we conclude $\neg req^{t''}$ for any $t'' \in]t' : t-1]$ with lemmas 3.4.1 and 3.4.2. Thus, $last_{req}(t) = t'$ holds and we can also conclude $last_{d\$.crd}(t) < t'$. Because of $\texttt{d\$.line\_fill.last}^{t}$ holds, lemma 3.4.4 guarantees acc_end^{t-1} and we conclude $next_{acc\_end}(t') = t-1$. The bus protocol according to equation (3.9) then guarantees the existence of some $f : \mathbb{Z}_{2^s} \rightarrow]t' : t[$ with

$$\begin{array}{l} f(2^s-1) = t-1 \wedge \forall k \in \mathbb{Z}_{2^s-1} : f(k) < f(k+1) \wedge \\ \forall k \in]t' : t-1[: reqp^k \wedge (brdy^k \iff k \in f(\mathbb{Z}_{2^s-1})) \wedge \\ \forall k \in \mathbb{Z}_{2^s} : badr^{f(k)+1} = 2^s \cdot \langle mem.adr^{t'}[a-1:s] \rangle + k \wedge \\ (\neg mem.mw^{t'} \implies mem.dout^{f(k)+1} = mem\_out^{f(k)+1}) \end{array}$$

Note that since $\texttt{d\$.line\_fill.req}^{t'}$ holds, we can apply lemma 3.4.1 in order to conclude $\neg\texttt{i\$.line\_fill}^{t'}$ and thus, we also have $mem.adr^{t'} =_s adr^{t'}$. We define a function $g : \mathbb{Z}_{2^s} \rightarrow]t' : t]$ by $g(i) := f(i) + 1$ for any $i \in \mathbb{Z}_{2^s}$. This function trivially fulfills $g(2^s-1) = t \wedge \forall i \in \mathbb{Z}_{2^s-1} : g(i) < g(i+1)$ and also $\forall i \in \mathbb{Z}_{2^s} : badr^{g(i)} = 2^s \cdot \langle adr^{last_{req}(t)}[a-1:s] \rangle + i$. Furthermore, $mem.mw^{t'}$ does not hold which leads to $mem.dout^{g(i)} = mem\_out^{g(i)} = mem^{g(i)}[badr^{g(i)}]$. We also know $brdy^{t''-1}$ iff $t'' \in g(\mathbb{Z}_{2^s-1})$ according to the above equation. This easily leads to $\texttt{d\$.line\_fill.mem}^{t''}$ for $t'' \in g(\mathbb{Z}_{2^s-1})$ and $\texttt{d\$.line\_fill.wait}^{t''}$ otherwise which concludes the claim. □

The following two lemmas state the according claim for a line write in the data cache and a line fill in the instruction cache, respectively. Since their proofs are almost identical to the one of the previous lemma, they are omitted in this thesis.

Lemma 3.4.12 *During a data cache line write, the physical memory is accessed in states* $\texttt{d\$.write\_back.mem}$ *and* $\texttt{d\$.write\_back.last}$*; there are exactly* 2^s *such accesses as specified by the bus protocol. Formally, we claim for any cycle* $t \in \mathbb{N}^+$*:*

$$\begin{array}{l} \texttt{d\$.write\_back.last}^{t} \implies \exists last_{req}(t) \wedge last_{d\$.crd}(t) < last_{req}(t) \wedge \\ \quad \texttt{d\$.write\_back.req}^{last_{req}(t)} \wedge \exists g : \mathbb{Z}_{2^s} \rightarrow]last_{req}(t) : t] : \\ \quad g(2^s-1) = t \wedge \forall i \in \mathbb{Z}_{2^s-1} : g(i) < g(i+1) \wedge \\ \quad \forall i \in \mathbb{Z}_{2^s} : badr^{g(i)} = 2^s \cdot \langle ev\_adr^{last_{req}(t)}[a-1:s] \rangle + i \wedge \\ \qquad mem.din^{g(i)} = d\$.dout^{g(i)} \wedge \\ \quad \forall t' \in]last_{req}(t) : t[: t' \in g(\mathbb{Z}_{2^s-1}) ? \\ \qquad \texttt{d\$.write\_back.mem}^{t'} : \texttt{d\$.write\_back.wait}^{t'} \end{array}$$

Lemma 3.4.13 *During an instruction cache line fill, the physical memory is accessed in states* `i$.line_fill.mem` *and* `i$.line_fill.last`*; there are exactly* 2^s *such accesses as specified by the bus protocol. Formally, we claim for any cycle* $t \in \mathbb{N}^+$*:*

$$
\begin{array}{l}
\texttt{i\$.line\_fill.last}^t \implies \exists last_{req}(t) \wedge last_{i\$.crd}(t) < last_{req}(t) \wedge \\
\quad \texttt{i\$.line\_fill.req}^{last_{req}(t)} \wedge \exists g : \mathbb{Z}_{2^s} \to]last_{req}(t) : t] : \\
\quad g(2^s - 1) = t \wedge \forall i \in \mathbb{Z}_{2^s - 1} : g(i) < g(i+1) \wedge \\
\quad \forall i \in \mathbb{Z}_{2^s} : badr^{g(i)} = (d\$.hit^{last_{req}(t)} \wedge d\$.dirty^{last_{crd}(t)})? \\
\qquad\quad 2^s \cdot \langle ev\_adr^{last_{req}(t)}[a-1:s]\rangle + i : \\
\qquad\quad 2^s \cdot \langle pc^{last_{req}(t)}[a-1:s]\rangle + i \wedge \\
\qquad i\$.din^{g(i)} = (d\$.hit^{last_{req}(t)} \wedge d\$.dirty^{last_{crd}(t)})? \\
\qquad\quad d\$.dout^{g(i)} : mem^{g(i)}[\langle badr^{g(i)}\rangle] \wedge \\
\quad \forall t' \in]last_{req}(t) : t[: t' \in g(\mathbb{Z}_{2^s - 1})? \\
\qquad \texttt{i\$.line\_fill.mem}^{t'} : \texttt{i\$.line\_fill.wait}^{t'}
\end{array}
$$

Note that the above lemma actually looks somewhat different from the previous two lemmas since it has to take both sources for an instruction cache line fill into account, the data cache and the physical memory. However, this does not affect the overall proof arguments, but just introduces an additional case split.

Lemma 3.4.14 *During a line fill in the data cache, any byte in any address in the current cache line is written exactly once with data read from the corresponding address in the physical memory. Formally, we use the* $[]_s$ *notation from definition 2.1.1 and for any* $t \in \mathbb{N}^+$*, we have:*

$$
\begin{array}{l}
\texttt{d\$.line\_fill.last}^t \implies \\
\quad \forall ad \in \left[d\$.adr^{last_{req}(t)}\right]_s \exists t' \in]last_{req}(t) : t] \exists i \in \mathbb{Z}_{2^s} : \\
\quad d\$.adr^{t'} = ad \wedge d\$.cdwb = 1^s \wedge badr^{t'} = \langle adr^{last_{req}(t)}[a-1:s]\rangle \cdot 2^s + i \wedge \\
\quad badr^{t'} = \langle ad\rangle \wedge mem.dout^{t'} = mem^{t'}[\langle ad\rangle] \wedge \\
\quad (\texttt{d\$.line\_fill.mem}^{t'} \vee \texttt{d\$.line\_fill.last}^{t'}) \wedge \\
\quad d\$.bcnt^{t'} = ad[s-1:0] \wedge \\
\quad \forall t'' \in]t' : t[: \langle d\$.bcnt^{t''}\rangle > \langle d\$.bcnt^{t'}\rangle \wedge \\
\quad \forall t'' \in]last_{req}(t) : t'[: (\texttt{d\$.line\_fill.mem}^{t''} \vee \texttt{d\$.line\_fill.last}^{t''}) \\
\qquad \implies \langle d\$.bcnt^{t''}\rangle < \langle d\$.bcnt^{t'}\rangle
\end{array}
$$

Proof: We apply lemma 3.4.11 in order to conclude

$$
\begin{array}{l}
\exists last_{req}(t) \wedge last_{d\$.crd}(t) < last_{req}(t) \wedge \\
\exists g : \mathbb{Z}_{2^s} \to]last_{req}(t) : t] : g(2^s - 1) = t \wedge \forall i \in \mathbb{Z}_{2^s - 1} : g(i) < g(i+1) \wedge \\
\forall i \in \mathbb{Z}_{2^s} : badr^{g(i)} = 2^s \cdot \langle adr^{last_{req}(t)}[a-1:s]\rangle + i \wedge \\
\quad mem.dout^{g(i)} = mem^{g(i)}[badr^{g(i)}] \wedge \\
\forall t' \in]last_{req}(t) : t[: t' \in g(\mathbb{Z}_{2^s - 1})? \, \texttt{d\$.line\_fill.mem}^{t'} : \texttt{d\$.line\_fill.wait}^{t'}
\end{array}
$$

Note that $d\$.bcnt.clr^{last_{req}(t)}$ and $d\$.bcnt.ce^{t'} \iff t' \in g(\mathbb{Z}_{2^s})$ holds for any $t' \in]last_{req}(t) : t]$. Thus, we conclude for any $i \in \mathbb{Z}_{2^s}$ that $\langle d\$.bcnt^{g(i)} \rangle = i$ which actually requires a separate induction proof in PVS. We choose $i := \langle ad[s-1:0] \rangle$ and $t' := g(i)$. For this cycle t', we trivially conclude $d\$.bcnt^{t'} = ad[s-1:0]$, $d\$.adr^{t'} = adr^{t'}[a-1:s] \cdot d\$.bcnt^{t'}$, and $d\$.cdwb^{t'} = 1^s$. Because of $ad \in \left[d\$.adr^{last_{req}(t)}\right]_s$, we also have $ad =_s adr^{last_{req}(t)}$. With corollary 3.4.7 for cycles $last_{req}(t)$ and t', we conclude $adr^{last_{req}(t)} = adr^{t'}$. We then have

$$\begin{aligned} badr^{t'} &= 2^s \cdot \langle adr^{last_{req}(t)}[a-1:s] \rangle + i \\ &= 2^s \cdot \langle ad[a-1:s] \rangle + \langle ad[s-1:0] \rangle \\ &= \langle ad \rangle \end{aligned}$$

and also

$$\begin{aligned} d\$.adr^{t'} &= adr^{t'}[a-1:s] \cdot d\$.bcnt^{t'} \\ &= adr^{last_{req}(t)}[a-1:s] \cdot \mathrm{bin}(i) \\ &= ad[a-1:s] \cdot ad[s-1:0] \\ &= ad. \end{aligned}$$

With $d\$.bcnt.ce^{t'} \iff t' \in g(\mathbb{Z}_{2^s})$ for any $j \in \mathbb{Z}_{2^s}$, we are also able to conclude the last two properties and finish the claim. □

For a line fill in the instruction cache and the writeback of a line in the data cache, there are once again similar lemmas where some states and signal names are substituted with almost identical proofs. Therefore, we only list the lemmas without proof for proper instantiation later on.

Lemma 3.4.15 *During a line write in the data cache, any byte in any address in the current cache line is written exactly once in the physical memory with data read from the corresponding address in the data cache. Formally, for any $t \in \mathbb{N}^+$, we have:*

$$\begin{aligned} &\texttt{d\$.write\_back.last}^t \implies \\ &\quad \forall ad \in \left[d\$.adr^{last_{req}(t)}\right]_s \exists t' \in]last_{req}(t) : t] \exists i \in \mathbb{Z}_{2^s} : \\ &\quad d\$.adr^{t'} = ad \wedge mem.mwb^{t'} = 1^s \wedge \\ &\quad badr^{t'} = \langle ev\_\, adr^{last_{req}(t)}[a-1:s] \rangle \cdot 2^s + i \wedge \\ &\quad badr^{t'} = \langle ad \rangle \wedge mem.din^{t'} = d\$.dout^{t'} \wedge \\ &\quad (\texttt{d\$.write\_back.mem}^{t'} \vee \texttt{d\$.write\_back.last}^{t'}) \wedge \\ &\quad d\$.bcnt^{t'} = ad[s-1:0] \wedge \\ &\quad \forall t'' \in]t' : t[: \langle d\$.bcnt^{t''} \rangle > \langle d\$.bcnt^{t'} \rangle \wedge \\ &\quad \forall t'' \in]last_{req}(t) : t'[: (\texttt{d\$.write\_back.mem}^{t''} \vee \texttt{d\$.write\_back.last}^{t''}) \\ &\qquad \implies \langle d\$.bcnt^{t''} \rangle < \langle d\$.bcnt^{t'} \rangle \end{aligned}$$

Lemma 3.4.16 *During a line fill in the instruction cache, any byte in any address in the current cache line is written exactly once with data read from the corresponding address in the physical memory or the data cache. Formally, for any $t \in \mathbb{N}^+$, we have:*

$$
\begin{array}{l}
\texttt{i\$.line\_fill.last}^t \implies \\
\quad \forall ad \in \left[i\$.adr^{last_{req}(t)}\right]_s \exists t' \in]last_{req}(t) : t] \exists i \in \mathbb{Z}_{2^s} : \\
\quad i\$.adr^{t'} = ad \wedge i\$.cdwb = 1^s \wedge badr^{t'} = \langle ad \rangle \wedge \\
\quad badr^{t'} = (d\$.hit^{last_{req}(t)} \wedge d\$.dirty^{last_{req}(t)})? \\
\qquad \langle ev\_adr^{last_{req}(t)}[a-1:s] \rangle \cdot 2^s + i : \\
\qquad \langle pc^{last_{req}(t)}[a-1:s] \rangle \cdot 2^s + i \wedge \\
\quad i\$.din^{t'} = (d\$.hit^{last_{req}(t)} \wedge d\$.dirty^{last_{req}(t)})?\, d\$.dout^{t'} : mem^{t'}[\langle ad \rangle] \wedge \\
\quad (\texttt{i\$.line\_fill.mem}^{t'} \vee \texttt{i\$.line\_fill.last}^{t'}) \wedge \\
\quad i\$.bcnt^{t'} = ad[s-1:0] \wedge \\
\quad \forall t'' \in]t' : t[: \langle i\$.bcnt^{t''} \rangle > \langle i\$.bcnt^{t'} \rangle \wedge \\
\quad \forall t'' \in]last_{req}(t) : t'[: (\texttt{i\$.line\_fill.mem}^{t''} \vee \texttt{i\$.line\_fill.last}^{t''}) \\
\qquad \implies \langle i\$.bcnt^{t''} \rangle < \langle i\$.bcnt^{t'} \rangle
\end{array}
$$

Corollary 3.4.17 *During a line fill in the data cache, the current content of the physical memory is written to the corresponding address in the data cache. Formally, we have $\forall t \in \mathbb{N}^+$:*

$$
\begin{array}{c}
\texttt{d\$.line\_fill.mem}^t \vee \texttt{d\$.line\_fill.last}^t \implies \\
\langle d\$.adr^t \rangle = badr^t \wedge d\$.din^t = mem^t[\langle d\$.adr^t \rangle]
\end{array}
$$

Proof: Let $\texttt{d\$.line\_fill.mem}^t \vee \texttt{d\$.line\_fill.last}^t$ hold. We apply lemma 3.1.2 and 3.1.3, respectively, in order to find some cycle t' with $\texttt{d\$.line\_fill.req}^{t'}$ and $\texttt{d\$.line\_fill.mem}^k \vee \texttt{d\$.line\_fill.wait}^k$ for any cycle $k \in]t' : t[$. With lemma 3.4.3, we additionally conclude $\neg busy^{t'}$ and $busy^k$ for any cycle $k \in]t' : t[$. Hence, we know $last_{req}(t) = t'$ and the bus protocol guarantees that the burst access started in cycle t' ends in some cycle $m > t'$ with $last_{req}(m) = t'$, $brdy^m \wedge \neg reqp^m$, and thus also $last_{req}(m+1) = t'$. With lemma 3.4.4, we conclude

$$\texttt{d\$.line\_fill.last}^{m+1} \vee \texttt{d\$.write\_back.last}^{m+1} \vee \texttt{i\$.line\_fill.last}^{m+1}$$

However, lemma 3.4.1 ensures $\neg\texttt{i\$.line\_fill}^{t'}$ and thus, we can use lemma 3.4.12 and 3.4.13 in order to conclude

$$\neg\texttt{d\$.write\_back.last}^{m+1} \wedge \neg\texttt{i\$.line\_fill.last}^{m+1}$$

which leads to $\texttt{d\$.line\_fill.last}^{m+1}$. Therefore, we conclude $m+1 \geq t$ and apply lemma 3.4.14 to cycle $m+1$ and address $d\$.adr^t$. Thus, we find a cycle $k' \in]t' : m+1]$ with $d\$.adr^{k'} = d\$.adr^t$, $d\$.adr^{k'} = badr^{k'}$, and $d\$.din^{k'} = mem^{k'}[\langle badr^{k'} \rangle]$. In other words, we only have to show $k' = t$ in

order to finish the claim. In case of $k' > t$, lemma 3.4.14 additionally guarantees $d\$.bcnt^{k'} > d\$.bcnt^t$ which is a contradiction to $d\$.adt^t = d\$.adr^{k'}$. Accordingly, $k' < t$ cannot hold since $d\$.bcnt^{k'} < d\$.bcnt^t$ is a contradiction to $d\$.adt^t = d\$.adr^{k'}$. This finishes the claim. □

Since the following two corollaries basically are basically just the line write and the instruction cache version of the previous one, their proofs are omitted in this thesis.

Corollary 3.4.18 *During a line write in the data cache, the current content of the data cache is written to the corresponding address in the physical memory. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned}&\texttt{d\$.write\_back.mem}^t \vee \texttt{d\$.write\_back.last}^t \implies\\ &\quad \langle d\$.adr^t \rangle = badr^t \wedge mem.din^t = d\$.dout^t\end{aligned}$$

Corollary 3.4.19 *During a line fill in the instruction cache, either the output of the data cache or the current content of the physical memory is written to the corresponding address in the instruction cache. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned}&\texttt{i\$.line\_fill.mem}^t \vee \texttt{i\$.line\_fill.last}^t \implies\\ &\quad \langle i\$.adr^t \rangle = badr^t \wedge\\ &\quad i\$.din^t = (d\$.hit^{last_{req}(t)} \wedge d\$.dirty^{last_{req}(t)})?\, d\$.dout^t : mem^t[\langle i\$.adr^t \rangle]\end{aligned}$$

Lemma 3.4.20 *The input of the data cache fulfills predicate* $valid\_input?$*.*

Proof: The first three items of predicate $valid\_input?$ are already concluded by lemma 3.4.10. Hence, we only consider the last three items, i.e., $\forall t \in \mathbb{N}^+$:

4. $d\$.tw^t \implies d\$.vw^t \wedge d\$.val\_in^t$

5. $d\$.vw^t \wedge d\$.val\_in^t \implies \exists j \in]last_{d\$.crd}(t) : t[: d\$.vw^j \wedge \neg d\$.val\_in^j \wedge$
   $(\forall k \in]last_{d\$.crd}(t) : j[: \neg vw^k) \wedge (\forall k \in]j : t[: \neg d\$.vw^k) \wedge$
   $(\forall ad \in \left[d\$.adr^t\right]_s, b \in \mathbb{Z}_B : \exists t' \in]j : t] : d\$.bw(ad,b)^{t'})$

6. $\exists l \in \mathbb{Z}_B : d\$.cdwb^t[l] \wedge \neg d\$.hit^{last_{d\$.crd}(t)} \implies$
   $\exists j \in]last_{d\$.crd}(t) : t[: d\$.vw^j \wedge \neg d\$.val\_in^j$

We will prove the three items separately.

4. The first item, $d\$.tw^t \implies d\$.vw^t \wedge d\$.val\_in^t$ immediately follows from equation (3.12).

5. Let $d\$.vw^t \wedge d\$.val\_in^t$ hold. We conclude $\texttt{d\$.line\_fill.last}^t$ and apply lemma 3.4.11 in order to obtain $\texttt{d\$.line\_fill.req}^{last_{d\$.req}(t)}$. We set $j := last_{d\$.req}(t)$. By lemma 3.4.11, we have $\neg d\$.vw^k$ for any

$k \in]j : t[$ since either $\texttt{d\$.line\_fill.mem}^k$ or $\texttt{d\$.line\_fill.last}^k$ holds. For cycles $k \in]last_{crd}(t) : j[$, we conclude from figure 3.5 and equation (3.12) that $d\$.vw^k$ can only hold in case of $\texttt{i\$.line\_fill}^k$ and $\texttt{d\$.write\_back.last}^k$. However, we then have $\texttt{d\$.idle}^{k+1}$ which is a contradiction to lemma 3.2.2 and thus, $\neg d\$.vw^k$ holds.

We conclude $d\$.adr^t =_s d\$.adr^{last_{req}(t)}$ by applying lemma 3.4.10 to cycles t and $last_{req}(t)$; thus, $\left[d\$.adr^t\right]_s = \left[d\$.adr^{last_{req}(t)}\right]_s$ trivially holds. With lemma 3.4.14, we find a cycle $t' \in]j : t[$ for address ad with $d\$.adr^{t'} = ad$ and $d\$.cdwb^{t'}[b]$, i.e., $d\$.bw(ad, b)^{t'}$. This concludes the proof of the second item.

6. Let $d\$.cdwb^t[l]$ hold for some $l \in \mathbb{Z}_B$ and $\neg d\$.hit^{last_{d\$.crd}(t)}$. It is sufficient to find some cycle $t' \in]last_{d\$.crd}(t) : t[$ with $\texttt{d\$.line\_fill.req}^{t'}$. From $d\$.cdwb^t[l]$, we conclude either $\texttt{d\$.\$write}^{t'}$, $\texttt{d\$.line\_fill.mem}^{t'}$, or $\texttt{d\$.line\_fill.last}^{t'}$. In the latter two cases, lemma 3.1.3 concludes the claim. Let therefore $\texttt{d\$.\$write}^{t'}$ hold. Because of $\neg d\$.hit^{last_{d\$.crd}(t)}$, we conclude $\texttt{d\$.line\_fill.last}^{t-1}$ and we apply lemma 3.1.3 to cycle $t-1$ in order to finish the claim. □

Lemma 3.4.21 *The input of the instruction cache fulfills valid_input?.*

The proof of this lemma is omitted in this thesis since it almost exactly matches the above proof for the data cache. Thus, since both input to instruction and data cache fulfill predicate *valid_input?*, both caches are also consistent.

3.4.2 Consistency invariant

As a next step, we introduce the main consistency invariant for the cache memory interface. This invariant basically consists of three parts, i.e., we have a partial invariant for each of the three memories. Since we want to show the cache memory interface to implement a memory interface, we claim the three memories to contain the content of the memory interface under certain conditions. Informally, both caches contain the memory interface content in case of a hit, while the physical memory content equals the memory interface content in case the data cache does not hold dirty data. Formally, we have:

$$\begin{aligned} &mif\_consistency := \\ &\quad (i\$.hit \implies i\$.dout = M_I[\langle i\$.adr^t\rangle]) \wedge \\ &\quad (d\$.hit \implies d\$.dout = M_I[\langle d\$.adr^t\rangle]) \wedge \\ &\quad (\neg(d\$.hit \wedge d\$.dirty) \implies \forall ad \in [d\$.adr]_s : \\ &\qquad\qquad mem[\langle ad\rangle] = M_I[\langle ad\rangle]) \end{aligned} \tag{3.17}$$

Note that for the following lemmas, we use the cache properties that our caches provide in addition to cache consistency according to definition 2.2.3. In particular, with these properties, it is easy to prove that

1. in state d$.write, the data cache signals a hit,
2. during a line fill, i.e., d$.line_fill or i$.line_fill, data or instruction cache, respectively, signal a miss,
3. in a line invalidation state, a cache signals a hit,
4. in state d$.write_back, the data cache signals a dirty hit.

We omit these proofs here since they are an immediate consequence of the properties according to definition 2.2.3 and the realization of the corresponding automata for instruction and data cache.

Lemma 3.4.22 *During a line fill in the instruction cache, there exists a cycle where the data cache is accessed on the address of the instruction cache line fill and either signalled a hit or was invalidated. Additionally, the data cache was not in state* d$.write *between the start of the instruction cache access and this cycle. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$
\begin{aligned}
&(\texttt{i\$.line\_fill.mem}^t \vee \texttt{i\$.line\_fill.last}^t) \implies \\
&\quad \exists t' \geq t : last_{i\$.crd}(t') = last_{i\$.crd}(t) \wedge \texttt{i\$.line\_fill.last}^{t'} \wedge \\
&\quad \exists k \in [last_{i\$.crd}(t) : t'] : d\$.adr^k =_s i\$.adr^t \wedge \\
&\qquad (\neg d\$.hit^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k) \wedge \\
&\qquad \forall k' \in [last_{i\$.crd}(t) : k] : \neg \texttt{d\$.\$write}^{k'} \wedge \\
&\quad \forall k' \in [last_{i\$.crd}(t) : t'] : \neg \texttt{d\$.line\_fill}^{k'}
\end{aligned}
$$

Proof: We set $l := last_{i\$.crd}(t)$. By the same arguments as in the proof of lemma 3.4.17, we find a cycle $t' \geq t$ with $\texttt{i\$.line\_fill.last}^{t'}$, $last_{req}(t) = last_{req}(t')$; i.e., in particular, $l = last_{i\$.crd}(t')$. With lemma 3.4.13, we conclude $\texttt{i\$.line\_fill.req}^{last_{req}(t)}$ and $\texttt{i\$.line\_fill}^{k'}$ holds for any $k' \in [last_{req}(t) : t']$. We therefore conclude $l = last_{req}(t) - 1$ and $\texttt{i\$.wait4dinit}^l$. In particular, this leads to $\texttt{d\$.idle}^l$ and lemma 3.4.1 additionally guarantees $\neg\texttt{d\$.line\_fill}^{k'}$ for any $k \in]l : t']$ which finishes the last proof obligation since $\texttt{d\$.idle}^l$ also holds. Note that since $i\$.cdwb^t = 1^B$ holds, item 3 of *valid_input*? additionally guarantees $i\$.adr^t =_s i\$.adr^l$. Since the data cache is snooped in cycle l, we also have $i\$.adr^l =_s d\$.adr^l$. We therefore only have to find a cycle k with the desired properties, i.e.,

$$
\begin{aligned}
&d\$.adr^k =_s d\$.adr^l \wedge (\neg d\$.hit^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k) \wedge \\
&\forall k' \in [l : k] : \neg \texttt{d\$.\$write}^{k'}
\end{aligned}
$$

If $\neg d\$.hit^l$ holds, the proof is trivially finished with $k := l$. We therefore assume $d\$.hit^l$ and split cases on $d\$.dirty^l$.

1. Let $\neg d\$.dirty^l$ hold. This leads to $\texttt{d\$.linv}^{l+1}$. Note that $i\$.adr^{l+1} =_s pc^{l+1}$ and $i\$.adr^l =_s pc^l$ hold and because of valid input according to

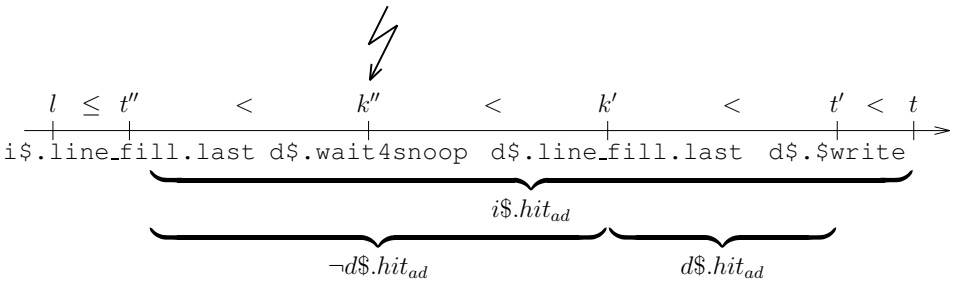

Figure 3.9: Correctness arguments for proof of lemma 3.4.23

definition 1.5.1, we also have $pc^l = pc^{l+1}$. Since $d\$.adr^{l+1} =_s i\$.adr^{l+1}$, $d\$.vw^{l+1}$, and $\neg d\$.val\_in^{l+1}$ also holds, this case is finished with $k := l + 1$.

2. Let $d\$.dirty^l$ hold. The instruction cache is then filled from the data cache, i.e., we have $\texttt{d\$.write\_back.last}^{t'}$, $\texttt{d\$.write\_back}^{k'}$ for any $k' \in]l : t'[$, $last_{d\$.crd}(t') = l$, and $ev\_adr^{t'} = d\$.adr^l$. The claim is concluded with $k := t'$ since both $d\$.vw^{t'} \wedge \neg d\$.val\_in^{t'}$ and $d\$.adr^{t'} =_s ev\_adr^{t'}$ hold. □

Lemma 3.4.23 *If the instruction cache signals a hit, the specification memory did not change since the cycle of the last write to the hit address. Formally, we have* $\forall t \in \mathbb{N}^+ \forall b \in \mathbb{Z}_B$*:*

$$i\$.hit^t \implies M_I^t[\langle i\$.adr^t\rangle] = M_I^{last_{i\$.bw(i\$.adr^t,b)}}[\langle i\$.adr^t\rangle]$$

Proof: Since we are going to argue bout six different cycles in this proof, figure 3.9 serves as a valuable illustration. Let $i\$.hit^t$ hold. We show the claim by contradiction, i.e., we set $l := last_{i\$.bw(i\$.adr^t,b)}$ and assume $M_I^t[\langle i\$.adr^t\rangle] \neq M_I^l[\langle i\$.adr^t\rangle]$. Hence, we find a cycle $t' \in [l : t[$ with $M_I.bw(i\$.adr^t, b)^{t'}$. In particular, $\texttt{d\$.\$write}^{t'}$, $i\$.adr^t = d\$.adr^{t'}$, as well as $d\$.hit^{t'}$ hold. Note that because of $i\$.bw(i\$.adr^t, b)^l$, $\texttt{i\$.line\_fill.mem}^l \vee \texttt{i\$.line\_fill.last}^l$ and $i\$.adr^t = i\$.adr^l$ also holds. Hence, we can apply lemma 3.4.22 to cycle l in order to find a cycle $t'' > l$ with $i\$.adr^{t''} =_s i\$.adr^t$, $\texttt{i\$.line\_fill.last}^{t''}$, $last_{i\$.crd}(t'') = last_{i\$.crd}(l)$ and a further cycle $k \in [last_{i\$.crd}(l) : t'']$ not explicitly listed in the illustration of figure 3.9 with

$$\begin{aligned} &d\$.adr^k =_s i\$.adr^l \wedge (\neg d\$.hit^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k) \wedge \\ &\quad \forall k' \in [last_{i\$.crd}(l) : k] : \texttt{¬d\$.\$write}^{k'} \\ &\quad \forall k' \in [last_{i\$.crd}(l) : t''] : \texttt{¬d\$.line\_fill}^{k'} \end{aligned}$$

In particular, this leads to $t' > k$ because of $\texttt{d\$.\$write}^{t'}$. Note also that because of $i\$.hit^t$, the continuous hit property, i.e., item 6 of definition 2.2.3,

for the instruction cache guarantees for some cycle $m \in [l : t[$

$$\begin{array}{l} \forall k' \in [l : t] : (i\$.adr^{k'} =_s i\$.adr^t \implies (i\$.hit^{k'} \iff k' \geq m)) \wedge \\ \forall k' \in [m : t[: \neg(vw^{k'} \wedge adr^{k'} =_s adr^t) \wedge (k' < m-1 \implies \neg tw^{k'}) \end{array} \tag{3.18}$$

Note that because of $i\$.tw^{t''} \wedge i\$.vw^{t''}$ and $i\$.adr^{t''} =_s i\$.adr^t$ we can conclude $m = t'' + 1$. In other words, the instruction cache was hit continuously on address $i\$.adr^t$ from cycle $t'' + 1$ on and therefore not invalidated on this address in the same interval which we will employ in constructing a contradiction.

Since $\neg d\$.hit^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k$ holds as well as $d\$.hit^{t'}$ and $d\$.adr^{t'} =_s d\$.adr^k$, we can use item 7 of definition 2.2.3 for cycles k and t' in order to find a cycle $k' \in [k : t'[$ with $d\$.tw^{k'} \wedge d\$.adr^{k'} =_s d\$.adr^{t'}$. In particular, $\texttt{d\$.line\_fill.last}^{k'}$ then holds and thus, $k' > t''$ since there can be no simultaneous line fill in both instruction and data cache. Now, we find an additional cycle $k'' < k'$ in the same cache access, i.e., $last_{d\$.crd}(k'') = last_{d\$.crd}(k')$ with $\texttt{d\$.wait4snoop}^{k''}$, $\texttt{d\$.line\_fill.req}^{k''+1}$, and $d\$.adr^{t'} =_s d\$.adr^{k''}$. Thus, the instruction cache is snooped in cycle k'' and we have $\neg\texttt{i\$.line\_fill}^{k''}$ and $i\$.adr^{k''} =_s d\$.adr^{t'}$; therefore, $k'' > t''$ holds as well. Hence, we can conclude $i\$.hit^{k''}$ with equation (3.18). This leads to $\texttt{i\$.linv}^{k''+1}$ or $\texttt{i\$.linv2}^{k''+1}$ and $i\$.vw^{k''+1} \wedge i\$.adr^{k''+1} =_s d\$.adr^{t'}$ which is a contradiction to equation (3.18) and thus finishes this proof. □

Lemma 3.4.24 *In a line fill in the instruction cache in cycle t, both the physical memory mem and the specification memory M_I are not altered from the beginning of the access on the instruction cache address of cycle t if there was no dirty hit in the data cache at the beginning of the access. Formally, we have $\forall t \in \mathbb{N}^+$:*

$$\begin{array}{l} (\texttt{i\$.line\_fill.mem}^t \vee \texttt{i\$.line\_fill.last}^t) \wedge \\ \quad \neg(d\$.hit^{last_{i\$.crd}(t)} \wedge d\$.dirty^{last_{i\$.crd}(t)}) \implies \\ \qquad M_I^t[\langle i\$.adr^t \rangle] = M_I^{last_{i\$.crd}(t)}[\langle i\$.adr^t \rangle] \wedge \\ \qquad mem^t[\langle i\$.adr^t \rangle] = mem^{last_{i\$.crd}(t)}[\langle i\$.adr^t \rangle] \end{array}$$

Proof: We set $l := last_{i\$.crd}(t)$. Let either $\neg d\$.hit^l$ or $\neg d\$.dirty^l$ hold and $\texttt{i\$.line\_fill.mem}^t \vee \texttt{i\$.line\_fill.last}^t$. By the same arguments as in the proof of lemma 3.4.17, we find a cycle $t' \geq t$ with $\texttt{i\$.line\_fill.last}^{t'}$, $\texttt{d\$.line\_fill}^k$ for any $k \in]l : t[$, and $last_{req}(t) = last_{req}(t')$; i.e., in particular, $l = last_{i\$.crd}(t')$. Since there was no dirty hit at the beginning of the cache access, the instruction cache reads its data from the memory, i.e., we conclude $\neg\texttt{d\$.write\_back}^k$ for any cycle $k \in [last_{i\$.crd}(t) : t'[$. In particular, $\neg mem.bw(ad, b)^k$ holds for *any* address ad and any byte b. Hence, we already have shown $mem^t[\langle i\$.adr^t \rangle] = mem^{last_{i\$.crd}(t)}[\langle i\$.adr^t \rangle]$.

We now assume $M_I^t[\langle i\$.adr^t \rangle] \neq M_I^{last_{i\$.crd}(t)}[\langle i\$.adr^t \rangle]$ and find a contradiction in order to finish the proof. We find a cycle $k \in [last_{i\$.crd}(t) : t'[$

with $M_I.bw(i\$.adr^t, b)^k$ for some $b \in \mathbb{Z}_B$. Note that $d\$.hit^k$, $\texttt{d\$.write}^k$, and $d\$.adr^k =_s i\$.adr^t$ hold in particular.

1. Let $\neg d\$.hit^l$ hold. With item 7 of definition 2.2.3 for cycles l and k, we then find a cycle $k' \in]l : k[$ with $d\$.tw^{l'}$, i.e., $\texttt{d\$.line\_fill.last}^{k'}$. This is a contradiction to lemma 3.4.1 because of $\texttt{i\$.line\_fill}^{k'}$ and thus finishes this case of the claim.

2. Let $d\$.hit^l$ hold. This leads to $\neg d\$.dirty^l$ and hence, $\texttt{d\$.linv}^{l+1}$, $d\$.adr^{l+1} =_s i\$.adr^t$, and $d\$.vw^{l+1} \wedge d\$.val\_in^{l+1}$. We can therefore apply item 7 of definition 2.2.3 for cycles $l+1$ and k in order to find a cycle k' and finish the case just like the above case. □

Lemma 3.4.25 *If the memory interface is consistent up to cycle t, the first implication of this consistency also holds in cycle $t+1$, i.e., $\forall t \in \mathbb{N}^+$:*

$$(\forall t' \in [1:t] : mif\_consistency^{t'}) \wedge i\$.hit^{t+1} \implies i\$.dout^{t+1} = M_I^{t+1}[\langle i\$.adr^{t+1}\rangle]$$

Proof: Let $i\$.hit^{t+1}$ hold. We have to show $i\$.dout^{t+1} = M_I^{t+1}[\langle i\$.adr^{t+1}\rangle]$ in order to finish the claim. We fix an arbitrary byte $b \in \mathbb{Z}_B$ and only show the claim for byte b of the above data word. Additionally, we use the shorthand notation $ad := d\$.adr^{t+1}$ in the following proof. Since the input of the instruction cache fulfills predicate $valid\_input?$, we can apply cache consistency from definition 1.5.3 in order to conclude

$$\left|i\$.dout^{t+1}\right|_b = \left|i\$.din^l\right|_b$$

for cycle $l := last_{i\$.bw(ad,b)}(t+1)$. Lemma 3.4.23 additionally guarantees $M_I^{t+1}[\langle ad\rangle] = M_I^l[\langle ad\rangle]$. Because of $i\$.bw(ad,b)^l$, we can also conclude $\texttt{i\$.line\_fill.mem}^l \vee \texttt{i\$.line\_fill.last}^l$. With lemma 3.4.19 for cycle l, we conclude $\langle i\$.adr^l\rangle = badr^l$ and also

$$i\$.din^l = (d\$.hit^{last_{req}(l)} \wedge d\$.dirty^{last_{req}(l)})?\, d\$.dout^l : mem^l[\langle i\$.adr^l\rangle]$$

and we only have to show $i\$.din^l = M_I^l[\langle ad\rangle]$. We split cases on $d\$.hit^{last_{req}(l)} \wedge d\$.dirty^{last_{req}(l)}$.

1. Let $d\$.hit^{last_{req}(l)} \wedge d\$.dirty^{last_{req}(l)}$ hold. This leads to $i\$.din^l = d\$.dout^l$, $\texttt{d\$.write\_back.mem}^l \vee \texttt{d\$.write\_back.last}^l$ and $d\$.hit^l \wedge d\$.dirty^l$. We use lemma 3.4.18 in order to conclude $\langle d\$.adr^l\rangle = badr^l$, i.e., $d\$.adr^l = i\$.adr^l$. Hence, $mif\_consistency$ for cycle l guarantees $d\$.dout^l = M_I^l[\langle i\$.adr^l\rangle]$ and this case is concluded.

2. Let $\neg(d\$.hit^{last_{req}(l)} \wedge d\$.dirty^{last_{req}(l)})$ hold. This leads to $i\$.din^l = mem^l[\langle i\$.adr^l\rangle]$. We can apply lemma 3.4.24 to cycle l in order to conclude

$$M_I^l[\langle i\$.adr^l\rangle] = M_I^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle] \wedge$$
$$mem^l[\langle i\$.adr^l\rangle] = mem^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle].$$

Putting all this together, we only have to show

$$M_I^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle] = mem^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle].$$

Since $i\$.cdwb^l = 1^B$ holds, item 3 of $valid\_input?$ guarantees $i\$.adr^l =_s i\$.adr^{last_{i\$.crd}(l)}$. With $mif\_consistency$ for cycle $last_{i\$.crd}(l)$, we know

$$mem^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle] = M_I^{last_{i\$.crd}(l)}[\langle i\$.adr^l\rangle]$$

and this lemma is finished. □

Lemma 3.4.26 *If the memory interface is consistent up to cycle t, the second implication of this consistency also holds in cycle $t+1$, i.e., $\forall t \in \mathbb{N}^+$:*

$$(\forall t' \in [1:t] : mif\_consistency^{t'}) \wedge d\$.hit^{t+1} \implies$$
$$d\$.dout^{t+1} = M_I^{t+1}[\langle d\$.adr^{t+1}\rangle]$$

Proof: Let $d\$.hit^{t+1}$ hold. We have to show $d\$.dout^{t+1} = M_I^{t+1}[\langle d\$.adr^{t+1}\rangle]$ in order to finish the claim. We fix an arbitrary but fixed byte $b \in \mathbb{Z}_B$ and only show the claim for byte b of the above data word. Additionally, we use the shorthand notation $ad := d\$.adr^{t+1}$ in the following proof.

1. Let there be some cycle $l \in [1:t]$ with

$$\left|d\$.dout^{t+1}\right|_b = \left|M_I^{l+1}[\langle ad\rangle]\right|_b \wedge \forall t' \in]l:t] : \neg d\$.bw(ad,b)^{t'}.$$

Hence, we only have to show $M_I^{l+1}[\langle ad\rangle] = M_I^{t+1}[\langle ad\rangle]$ in order to conclude the claim. We therefore assume inequality and have to find a contradiction. Because of $M_I^{l+1}[\langle ad\rangle] \neq M_I^{t+1}[\langle ad\rangle]$, we find some cycle $t' \in [l+1:t]$ with $M_I.bw(ad,b)^{t'}$. Since this implies $d\$.bw(ad,b)^{t'}$, we already have a contradiction and the case is finished.

2. We only have to find a cycle $l \in [1:t]$ with

$$\left|d\$.dout^{t+1}\right|_b = \left|M_I^{l+1}[\langle ad\rangle]\right|_b \wedge \forall t' \in]l:t] : \neg d\$.bw(ad,b)^{t'}.$$

in order to finish the overall claim. Since the input of the data cache fulfills predicate $valid\_input?$ according to lemma 3.4.20, we can apply cache consistency from definition 1.5.3 in order to conclude

$$\left|d\$.dout^{t+1}\right|_b = \left|d\$.din^l\right|_b$$

for cycle $l := last_{d\$.bw(ad,b)}(t+1)$. This cycle l fulfills $\forall t' \in]l : t] : \neg d\$.bw(ad,b)^{t'}$ according to proposition 1.2.9. Hence, we only have to show $\left|d\$.din^l\right|_b = \left|M_I^{l+1}[\langle ad\rangle]\right|_b$ in order to finish the claim. Since $d\$.bw(ad,b)^l$ holds, we know

$$\texttt{d\$.line\_fill.mem}^l \vee \texttt{d\$.line\_fill.last}^l \vee \texttt{d\$.\$write}^l.$$

in addition to $ad = d\$.adr^l$. We finally split cases on the possible states in cycle l.

(a) Let $\texttt{d\$.\$write}^l$ hold. The specification memory M_I is then updated on byte b of address ad with the current data input in cycle l, i.e., we have $M_I.bw(ad,b)^l$. Hence, we conclude $d\$.din^l = din^l$ and $\left|M_I^{l+1}[\langle ad\rangle]\right|_b = \left|d\$.din^l\right|_b$ which finishes this case of the claim.

(b) Let $\texttt{d\$.line\_fill.mem}^l \vee \texttt{d\$.line\_fill.last}^l$ hold. This leads to $M_I^{l+1}[\langle ad\rangle] = M_I^l[\langle ad\rangle]$ and lemma 3.4.17 additionally guarantees $d\$.din = mem^l[\langle ad\rangle]$. Since $d\$.hit^l$ does not hold, we can use $mif\_consistency$ in cycle l in order to conclude $mem^l[\langle ad\rangle] = M_I^l[\langle ad\rangle]$ and thus finish the claim. □

Lemma 3.4.27 *If the data cache does not signal a dirty hit in some cycle, but was in state* d$.$write *in a previous cycle for an address in the same cache line, this line was written back in between. Formally, we have* $\forall t \in \mathbb{N}^+ \forall t' \in [1 : t[$ *:*

$$\begin{aligned}&\texttt{d\$.\$write}^{t'} \wedge d\$.adr^t =_s d\$.adr^{t'} \wedge (\neg d\$.hit^t \vee \neg d\$.dirty^t) \implies\\ &\quad \exists m \in]t' : t[: \texttt{d\$.write\_back.last}^m \wedge last_{d\$.crd}(m) > t' \wedge\\ &\qquad d\$.adr^m =_s d\$.adr^t\end{aligned}$$

Proof: Let $\texttt{d\$.\$write}^{t'}$, $d\$.adr^t =_s d\$.adr^{t'}$ and $\neg d\$.hit^t \vee \neg d\$.dirty^t$ hold. Because of $\texttt{d\$.write}^{t'}$, we conclude $d\$.hit^{t'}$, $d\$.dw^{t'}$, and $d\$.dty^{t'}$. Hence, we can use the consistency of the dirty bit, i.e., item 4 of definition 2.2.3, in order to find a cycle $k \in]t' : t[$ with

$$\begin{aligned}&\left(d\$.dw^k \wedge \neg d\$.dty^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k\right) \wedge\\ &\left(last_{d\$.crd}(k) \leq t' \vee d\$.dirty^{last_{d\$.crd}(k)} \wedge\right.\\ &\left.d\$.adr^t =_s d\$.hit^{last_{d\$.crd}(k)} ? \, d\$.adr^k : d\$.ev^{last_{d\$.crd}(k)}\right).\end{aligned}$$

Note that $\texttt{d\$.idle}^{t'+1}$ holds and thus, $last_{d\$.crd}(k) > t'$. Hence, the above equation simplifies to

$$\begin{aligned}&\left(d\$.dw^k \wedge \neg d\$.dty^k \vee d\$.vw^k \wedge \neg d\$.val\_in^k\right) \wedge d\$.dirty^{last_{d\$.crd}(k)} \wedge\\ &d\$.adr^t =_s d\$.hit^{last_{d\$.crd}(k)} ? \, d\$.adr^k : d\$.ev^{last_{d\$.crd}(k)}.\end{aligned}$$

In cycle k, the dirty cache line is either invalidated or marked as clean. Hence, cycle k is either a line invalidation, a line fill, or the end of a line write that was initiated by a snoop access of the instruction cache. Furthermore, note that $\texttt{d\$.linv}^k$ cannot hold since this would imply $last_{d\$.crd}(k) = k-1$ and $\neg d\$.dirty^{k-1}$ which is a contradiction to the above equation.

1. Let $hit^{last_{d\$.crd}(k)} \wedge d\$.adr^t =_s d\$.adr^k$ hold. This easily leads to $\neg\texttt{d\$.line\_fill}^k$ since a line fill only occurs after a miss. Therefore, cycle k was during the writeback of a dirty cache line, i.e., $d\$.dw^k \wedge \neg d\$.dty^k$ or $d\$.vw^k \wedge \neg d\$.val\_in^k$ leads to $\texttt{d\$.write\_back.last}^k$. This case is finished with $m := k$.

2. Let $\neg d\$.hit^{last_{d\$.crd}(k)} \wedge d\$.ev^{last_{d\$.crd}(k)} =_s d\$.adr^t$ hold. Because of the miss, cycle k was not during the writeback of a dirty line and $\texttt{d\$.line\_fill}^k$ holds. Since the miss was dirty, however, we find a cycle $k' < k$ in the same cache access, i.e., $last_{crd}(k) = last_{crd}(k')$, where the dirty cache line was written back, i.e., $\texttt{d\$.write\_back.last}^{k'}$. Since $d\$.adr^{k'} =_s ev\_adr^{k'} = d\$.ev^{last_{d\$.crd}(k)}$ also holds, the proof is finished with $m := k'$. □

Lemma 3.4.28 *The cache memory interface fulfills the consistency invariant, i.e.,* $\forall t \in \mathbb{N}^+ : mif\_consistency^t$.

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 1}$): In the initial cycle, both caches are empty and cannot signal a hit; thus, the first two implications are trivially fulfilled. Since we define the initial content of the memory interface M_I as the initial content of the physical memory mem, the third implication is also fulfilled and the induction base is concluded.
Induction step ($\mathbf{t \to t+1}$): Let the consistency invariant hold up to cycle t. Lemmas 3.4.25 and 3.4.26 then show the first two implications of the consistency invariant for cycle $t+1$. Hence, we only have to consider the third implication. Let therefore $\neg(d\$.hit^{t+1} \wedge d\$.dirty^{t+1})$ hold. We fix an arbitrary address $ad \in \left[d\$.adr^{t+1}\right]_s$ and we only have to show $\left|mem^{t+1}[\langle ad\rangle]\right|_b = \left|M_I^{t+1}[\langle ad\rangle]\right|_b$. Since we once again have to argue about multiple cycle, figure 3.10 illustrates the proof.

1. Let there be some cycle $l \in [1:t]$ with

$$\left|mem^{t+1}[\langle ad\rangle]\right|_b = \left|M_I^l[\langle ad\rangle]\right|_b \wedge \forall t' \in]l:t] : \neg mem.bw(ad,b)^{t'}.$$

   Hence, we only have to show $\left|M_I^l[\langle ad\rangle]\right|_b = \left|M_I^{t+1}[\langle ad\rangle]\right|_b$ which we do by contradiction. Let $\left|M_I^l[\langle ad\rangle]\right|_b \neq \left|M_I^{t+1}[\langle ad\rangle]\right|_b$ hold. We then find some cycle before $t+1$ where byte b of address ad was updated in M_I, i.e., a cycle $t' \in [l:t]$ with $M_I.bw(ad,b)^{t'}$. Since $\texttt{d\$.\$write}^{t'}$ and

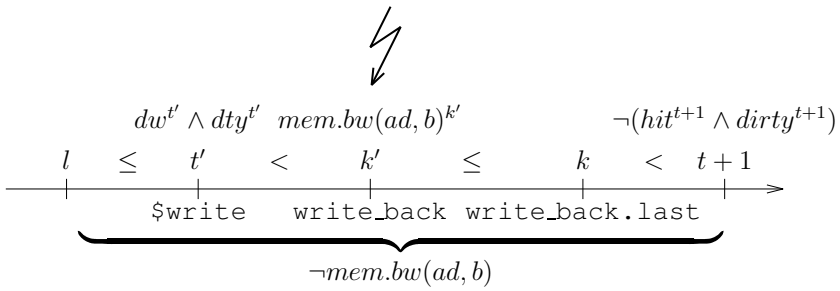

Figure 3.10: Correctness arguments for proof of lemma 3.4.28

$d\$.adr^{t'} = adr^{t'} = ad$ hold while in cycle $t+1$, the cache line is either no longer hit or no longer dirty, we can apply lemma 3.4.27 in order to find some cycle $k \in]t' : t]$ with

$$\texttt{d\$.write\_back.last}^k \wedge last_{d\$.crd}(k) > t' \wedge d\$.adr^k =_s d\$.adr^{t+1}.$$

This is illustrated in figure 3.10. The line write ending in cycle k is complete, i.e., any address in the same line is also written back to the physical memory. With lemma 3.4.15 for cycle k, we therefore find some cycle $k' \in]last_{req}(k) : k]$ with $mem.bw(ad,b)^{k'}$ which is a contradiction and thus finishes this case of the claim.

2. Let there be no cycle $l \in [1 : t]$ with

$$\left|mem^{t+1}[\langle ad \rangle]\right|_b = \left|M_I^l[\langle ad \rangle]\right|_b \wedge \forall t' \in]l : t] : \neg mem.bw(ad,b)^{t'}.$$

We show that this case cannot occur by contradiction. We split cases on $\exists^{last}_{mem.bw(ad,b)}(t+1)$.

   (a) Let $\exists^{last}_{mem.bw(ad,b)}(t+1)$ hold. We set $l := last_{mem.bw(ad,b)}(t+1)$ and lemma 3.1.1 guarantees $\left|mem^{t+1}[\langle ad \rangle]\right|_b = \left|mem.din^l\right|_b$. We trivially know $\forall t' \in]l : t] : \neg mem.bw(ad,b)^{t'}$. Thus, we only have to show $\left|M_I^l[\langle ad \rangle]\right|_b = \left|mem.din^l\right|_b$ in order to find a contradiction. Because of $mem.bw(ad,b)^l$, we conclude $ad = badr^l$ and

$$\texttt{d\$.write\_back.mem}^l \vee \texttt{d\$.write\_back.last}^l$$

   and lemma 3.4.18 guarantees $\langle d\$.adr^l \rangle = badr^l$ and $mem.din^l = d\$.dout^l$. Because of $d\$.hit^l$, $mif\_consistency$ for cycle l guarantees $d\$.dout^l = M_I^l[\langle d\$.adr^l \rangle]$ which finishes this case of the claim.

   (b) Let $\neg\exists^{last}_{mem.bw(ad,b)}(t+1)$ hold. We set $l := 1$ and trivially know $mem^{t+1}[\langle ad \rangle] = init\_mem[\langle ad \rangle]$ and $M_I^l = init\_mem$. Since $mem.bw(ad,b)^{t'}$ cannot hold for any $t' \in]l : t]$ we have a contradiction and the claim is concluded. □

3.4.3 Correct memory interface

In order to conclude the consistency parts from definition 1.5.2 of a correct memory interface, we basically just have to show that forwarding in the cache memory interface is correct.

Lemma 3.4.29 *During a line fill in the data cache automaton, the address given by adr is read from the physical memory and written into the forwarding register exactly once. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\begin{aligned}
&\texttt{d\$.line_fill.last}^t \implies \\
&\quad \exists t' \in]last_{req}(t) : t] : adr^{t'}[s-1:0] = d\$.bcnt^{t'} \wedge \\
&\quad (\texttt{d\$.line_fill.mem}^{t'} \vee \texttt{d\$.line_fill.last}^{t'}) \wedge \\
&\quad d\$.fwd^{t'+1} = M_I^{t'}[\langle adr^{t'} \rangle] \wedge \\
&\quad \forall k \in]t' : t] : adr^k[s-1:0] \neq \wedge d\$.bcnt^k \wedge \\
&\qquad \neg \texttt{d\$.wirte_back}^k \wedge \neg d\$.snoop^k
\end{aligned}$$

Proof: We set $l := last_{req}(t)$ and apply lemma 3.4.11 in order to conclude $\texttt{d\$.line\_fill.req}^l$ and $\texttt{d\$.line\_fill}^k$ for any $k \in [l : t]$. With equation (3.13), this leads to $d\$.adr^l =_s adr^l$ and because of corollary 3.4.7, $adr^t =_s d\$.adr^l$ holds. Hence, we can apply lemma 3.4.14 to cycle t in order to find a cycle t' for address $adr^t \in \left[d\$.adr^l\right]_s$ with $d\$.bcnt^{t'} = adr^t[s-1:0]$ and $d\$.adr^{t'} = adr^t$; additionally, the data cache automaton is either in state $\texttt{d\$.line\_fill.mem}$ or $\texttt{d\$.line\_fill.last}$ in cycle t', $mem.dout^{t'} = mem^{t'}[\langle adr^t \rangle]$, and for any later cycle $t'' \in]t' : t[$, we know $\langle d\$.bcnt^{t''} \rangle > \langle d\$.bcnt^{t'} \rangle$, i.e., in particular, $d\$.bcnt^{t''} \neq d\$.bcnt^{t'}$.

Since cycle t' is in a line fill, we know $\neg d\$.hit^{t'}$. With lemma 3.4.28, we have $mif\_consistency$ in cycle t' and thus, we conclude $mem^{t'}[\langle d\$.adr^{t'} \rangle] = M_I^{t'}[\langle d\$.adr^{t'} \rangle]$. Hence, we have $d\$.fwd\_in^{t'} = M_I^{t'}[\langle adr^t \rangle]$ which leads to $d\$.fwd^{t'+1} = M_I^{t'}[\langle adr^t \rangle]$. Hence, only the claim about the cycles between t' and t remains to show. We therefore fix some cycle $k \in]t' : t]$ and have to show

$$adr^k[s-1:0] \neq \wedge d\$.bcnt^k \wedge \neg \texttt{d\$.wirte\_back}^k \wedge \neg d\$.snoop^k.$$

With the above observations, we already know $\texttt{d\$.line\_fill}^k$ and thus, $\neg \texttt{d\$.wirte\_back}^k$ and $\neg d\$.snoop^k$. Since $\langle d\$.bcnt^k \rangle > \langle d\$.bcnt^{t''} \rangle$ also holds according to the above observations, the claim is finished because of $d\$.bcnt^{t'} = adr^t[s-1:0]$. □

Lemma 3.4.30 *In the cycle after a line fill in the data cache automaton, the forwarding register contains the data to be read from the cache memory interface. Formally, we have* $\forall t \in \mathbb{N}^+$*:*

$$\texttt{d\$.line\_fill.last}^t \implies d\$.fwd^{t+1} = M_I^t[\langle adr^t \rangle]$$

Proof: With lemma 3.4.29, we find a cycle $t' \in]last_{req}(t) : t]$ with

$$d\$.fwd^{t'+1} = M_I^{t'}[\langle adr^{t'}\rangle] \wedge$$
$$\forall k \in]t' : t] : adr^k[s-1:0] \neq d\$.bcnt^k \wedge \neg \texttt{d\$.wirte\_back}^k \wedge \neg d\$.snoop^k$$

With corollary 3.4.7 for cycle t' and t, we conclude $adr^t = adr^{t'}$. Since M_I is only updated in state d\$.\$write, and for any cycle k, $t' \leq k \leq t$, d\$.line_fill^k holds, we know $M_I^t = M_I^{t'}$. Thus, we know $M_I^t[\langle adr^t\rangle] = M_I^{t'}[\langle adr^{t'}\rangle]$ and only have to show $d\$.fwd^{t'+1} = d\$.fwd^{t+1}$ in order to finish the claim. According to equation (3.14), $d\$.fwd$ is not changed unless $d\$.sw \wedge eq_s(d\$.bcnt, adr\_d[s-1:0])$ with $adr\_d =$ d\$.wirte_back$?\, ev_adr : adr$ holds. We fix a cycle $k \in]t' : t]$ and conclude $adr\_d^k = adr^k$ because of $\neg$d\$.wirte_back^k; hence, $adr^k[s-1:0] \neq d\$.bcnt^k$ ensures that $d\$.fwd$ is not updated in cycle k and the claim is finished. □

Lemma 3.4.31 *The data port of the cache memory interface is consistent, i.e., we have* $\forall t \in \mathbb{N}^+$*:*

$$mr^t \wedge \neg dbusy^t \implies dout^t = M_I^t[\langle adr^t\rangle]$$

Proof: Let $mr^t \wedge \neg dbusy^t$ hold. We split cases on the possible states of the data cache FSD where this situation can occur. Note that we can trivially exclude d\$.\$write^t since mw^t holds in this state which is a contradiction to mr^t.

1. Let d\$.idle^t hold. We then know $d\$.hit^t$, $d\$.adr^t = adr^t$, and $dout^t = d\$.dout^t$. The claim $dout^t = M_I^t[\langle adr^t\rangle]$ then follows immediately from $mif\_consistency$ which is guaranteed by lemma 3.4.28.

2. Let d\$.line_fill.last^t hold. We apply lemma 3.4.30 to cycle t in order to conclude $d\$.fwd^{t+1} = M_I^t[\langle adr^t\rangle]$. With equation (3.14) and (3.15), we have $d\$.fwd^{t+1} = d\$.fwd\_in^t = dout^t$ and thus, the claim is concluded. □

Lemma 3.4.32 *The instruction port of the cache memory interface is consistent, i.e., we have* $\forall t \in \mathbb{N}^+$*:*

$$imr^t \wedge \neg ibusy^t \implies inst^t = M_I^t[\langle pc^t\rangle]$$

Since this proof for the instruction access port uses exactly the same arguments as the above proof for the data access port, we omit it in this thesis. After thus finishing consistency of the cache memory interface, we finally have to prove liveness according to definition 1.5.2 of a correct memory interface, i.e., any of the two busy signals becomes inactive eventually. With bus protocol liveness as an assumption, we basically only have to focus on absence of deadlock situations. Such a situation, e.g., could arise in case of simultaneous snoop accesses since both data and instruction cache are blocking.

Proposition 3.4.33 *In the cycle after an inactive dbusy, the data cache automaton is in state* `idle`*, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\neg dbusy^t \implies \texttt{d\$.idle}^{t+1}$$

Proof: The proof of this proposition follows immediately from figure 3.5 and equation (3.12). □

Lemma 3.4.34 *If the data cache is not in state* `idle`*, it is sufficient to show liveness of the data access port from the start of the cache access, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\neg \texttt{d\$.idle}^{t+1} \wedge \exists^{next}_{\neg dbusy}(last_{d\$.crd}(t+1)) \implies \exists^{next}_{\neg dbusy}(t)$$

Proof: Let $\neg \texttt{d\$.idle}^{t+1}$ hold and $\neg dbusy^{t'}$ for some $t' \geq last_{d\$.crd}(t+1)$. Since the claim is finished if $t' \geq t$ also holds, we can assume $t' < t$. We apply lemma 3.2.2 to cycle $t+1$ in order to conclude $\neg \texttt{d\$.idle}^{t'+1}$ since $t'+1 \in]last_{d\$.crd}(t+1) : t+1]$ holds. With lemma 3.4.33, this leads to $dbusy^{t'}$ which is a contradiction and finishes the lemma. □

Corollary 3.4.35 *It is sufficient to show liveness of the data access port from the start of the cache access, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\exists^{next}_{\neg dbusy}(last_{d\$.crd}(t+1)) \implies \exists^{next}_{\neg dbusy}(t)$$

Proof: Because of lemma 3.4.34, we only have to consider $\texttt{d\$.idle}^{t+1}$. If $last_{d\$.crd}(t+1) = t$ holds, the claim is trivially fulfilled. Let therefore $last_{d\$.crd}(t+1) = last_{d\$.crd}(t)$ hold. Since the claim is finished if $\neg dbusy^t$ holds, we can additionally assume $dbusy^t$. Because of $\neg d\$.crd$, this leads to $\neg \texttt{d\$.idle}^t$. Since $\texttt{d\$.idle}^1$ trivially holds, we know $t > 1$ and can apply lemma 3.4.34 to cycle $t-1$ in order to finish the claim. □

Lemma 3.4.36 *The instruction cache FSD eventually reaches an initial state, i.e.,* $\forall t \in \mathbb{N}^+$*:*

$$\exists t' \geq t : \texttt{i\$.idle}^{t'} \vee \texttt{i\$.wait4dinit}^{t'}$$

Proof: Let $\texttt{i\$.line\_fill}^t \vee \texttt{i\$.linv}^t \vee \texttt{i\$.linv2}^t$ hold for otherwise, the claim is finished with $t' := t$.

1. Let $\texttt{i\$.line\_fill}^t$ hold. With lemma 3.1.2 or lemma 3.1.3, we find a cycle $k < t$ with $\texttt{i\$.line\_fill.req}^k$ and $\texttt{i\$.line\_fill}^{k'}$ for any cycle $k' \in]k : t[$. We can apply lemma 3.4.3 in order to conclude $\neg busy^k$, i.e., $request^k$ holds. Liveness of the bus protocol and the physical memory according to equation (3.9) then guarantees that there exists some $l \geq k$ with acc_end^l, i.e., $\texttt{i\$.line\_fill.last}^{l+1}$ and $\texttt{i\$.idle}^{l+2}$. We finish this case with $t' := l+2$.

2. Let $\texttt{i\$.linv}^t$ or $\texttt{i\$.linv2}^t$ hold. We trivially conclude $\texttt{i\$.idle}^{t+1}$ or $\texttt{i\$.wait4dinit}^{t+1}$ and the claim is finished with $t' := t + 1$. □

Lemma 3.4.37 *If the data cache automaton is in state* `d$.idle` *while the instruction cache automaton is not in state* `i$.wait4dinit`*, the data cache access is live. Formally,* $\forall t \in \mathbb{N}^+$*:*

$$\texttt{d\$.idle}^t \wedge \neg \texttt{i\$.wait4dinit}^t \implies \exists^{next}_{\neg dbusy}(t)$$

Proof: Since $\neg\texttt{i\$.wait4dinit}^t$ holds, the data cache is not snooped in cycle t. In case of a read or write hit in cycle t, $\neg dbusy^t$ or $\neg dbusy^{t+1}$ hold, respectively. Thus, we only have to consider misses, i.e., $\neg d\$.hit^t$. The remaining arguments are based on figure 3.5 and equations (3.10) and (3.12).

1. Let a clean miss occur in cycle t, i.e., $\texttt{d\$.wait4snoop}^{t+1}$ holds. State `d$.wait4snoop` is left as soon as the instruction cache reaches an initial state which happens eventually according to lemma 3.4.36. Thus, we find a cycle $t' > t + 1$ with $\texttt{d\$.line\_fill.req}^{t'}$. According to lemma 3.4.3, $\neg busy^{t'}$ holds and thus, liveness of the bus protocol guarantees some cycle $t'' > t'$ with $acc\_end^{t''}$, i.e., $brdy^{t''} \wedge \neg reqp^{t''}$. Therefore, $\texttt{d\$.line\_fill.last}^{t''+1}$ holds. In case of a read access, $\neg dbusy^{t''+1}$ also holds, while a write access takes on more cycle, i.e., $\neg dbusy^{t''+2}$ holds which finishes this case of the claim.

2. Let a dirty miss occur in cycle t, i.e., $\texttt{d\$.wait4mem}^{t+1}$ holds. State `d$.wait4mem` is left as soon as the instruction cache FSD leaves state `i$.line_fill` which happens eventually because of the liveness of the bus protocol. Thus, $\texttt{d\$.write\_back.req}^{t'}$ holds for some $t' > t + 1$. By the same arguments as in the above case for `d$.line_fill`, we find a cycle $t'' > t'$ with $\texttt{d\$.write\_back.last}^{t''}$ and $last_{d\$.crd}(t'') = t$. Lemma 3.4.2 guarantees that $\texttt{i\$.line\_fill}^{t''}$ does not hold since this leads to the contradiction $d\$.hit^t$. Hence, we have $\texttt{d\$.wait4snoop}^{t''+1}$ and finish the claim as in the above case. □

Lemma 3.4.38 *If the data cache automaton is in state* `d$.idle`*, the data cache access is live. Formally,* $\forall t \in \mathbb{N}^+$*:*

$$\texttt{d\$.idle}^t \implies \exists^{next}_{\neg dbusy}(t)$$

Proof: Let $\texttt{d\$.idle}^t$ hold. If there exists some cycle $t' \geq t$ with $\texttt{d\$.idle}^{t'} \wedge \neg\texttt{i\$.wait4dinit}^{t'}$, we can apply lemma 3.4.37 to cycle t' in order to conclude the claim. We therefore assume that there exists *no* cycle $t' \geq t$ with $\texttt{d\$.idle}^{t'} \wedge \neg\texttt{i\$.wait4dinit}^{t'}$. In particular, $\texttt{i\$.wait4dinit}^t$ holds. Hence, the data cache is snooped in cycle t and $\texttt{i\$.line\_fill.req}^{t+1}$ holds. In

case of a snoop miss, we have a contradiction because both $\texttt{d\$.idle}^{t+1}$ and $\neg\texttt{i\$.wait4dinit}^{t+1}$ holds. Hence, we can assume $d\$.hit^t$.

For a clean hit, we have $\texttt{d\$.snoop}^{t+1}$ and $\texttt{d\$.idle}^{t+2}$ which is also a contradiction because $\texttt{i\$.line\_fill.wait}^{t+2}$ holds. Thus, we can assume a dirty hit, i.e., $\texttt{d\$.write\_back.req}^{t+1}$ and $\texttt{i\$.line\_fill.req}^{t+1}$. In this case, lemma 3.4.3 together with the bus protocol guarantees that there exists some cycle $t' > t+1$ with $\texttt{d\$.write\_back.last}^{t'}$ and $last_{d\$.crd}(t') = t$; note that $\texttt{i\$.write\_back.last}^{t'}$ also holds. Hence, $\texttt{d\$.idle}^{t'+1}$ and $\texttt{i\$.idle}^{t'+1}$ both hold which is a contradiction and the claim is finished. □

Lemma 3.4.39 *The data port of the cache memory interface is live, i.e.,* $\forall t \in \mathbb{N}^+ : \exists^{next}_{\neg dbusy}(t)$

Proof: Because of lemma 3.4.38 for cycle t, we only have to consider the case $\neg\texttt{d\$.idle}^t$. This easily leads to $last_{d\$.crd}(t+1) = last_{d\$.crd}(t)$. With lemma 3.2.2, we trivially conclude $\texttt{d\$.idle}^{last_{d\$.crd}(t+1)}$ and $last_{d\$.crd}(t+1) > 0$. We apply lemma 3.4.38 to cycle $last_{d\$.crd}(t+1)$ in order to obtain $\exists^{next}_{\neg dbusy}(last_{d\$.crd}(t+1))$. Lemma 3.4.35 then concludes the claim. □

Lemma 3.4.40 *The instruction port of the cache memory interface is live, i.e.,* $\forall t \in \mathbb{N}^+ : \exists^{next}_{\neg ibusy}(t)$

The liveness proof for the instruction access port of the cache memory interface is omitted in this thesis due to its similarity to the presented proof for the data port.

Theorem 3.4.41 *The cache memory interface implements a correct memory interface according to definition 1.5.2.*

Proof: The claim immediately follows from lemmas 3.4.31, 3.4.32, 3.4.39, and 3.4.40. □

Thus, we have formally verified a cache memory interface to implement a correct memory interface based on the cache properties proved in the last chapter. Hence, we can, e.g., plug in instruction and data caches with different associativity together with their correctness proof from the previous chapter without affecting a single argument in these proofs. This yields a correct implementation on the gate level down to the level of a bus protocol.

3.5 Related work

In [RMK03], Roychoudhury formalizes a bus protocol *without* a memory in the model checker SMV. He uses a fixed burst length of two; in addition, since the modeling is only on the protocol level, there is no way to express that the second data of the burst actually belongs to the incremented memory

address, i.e., a later read from this incremented address returns the data previously written in a burst. His main focus is on bus arbitration and absence of deadlock situations and he does not consider verifying an implementation of the protocol. Amjad [Amj04], on the other hand, is close to an actual bit-level implementation of the protocol with a fixed burst length of four; however, in his verification, he also ignores data consistency due to state-space explosion in model-checking. Schmaltz [SB03], finally, considers the same protocol in the theorem prover ACL2. He focuses on data consistency, but does not consider bursts. In summary, a formal model of a bus protocol with bursts and a memory and the verification of its implementation is not reported; previous approaches focused only on the verification of some of these aspects, while leaving it open how to close the remaining gaps.

There are various formal proofs of correctness of cache coherence protocols using either model checking or theorem proving. Park and Dill, e.g., verify the FLASH cache coherence protocol using the theorem prover PVS [PD96]. McMillan verifies the same protocol using his model checker SMV [McM01] and even achieves the verification of a parameterized version of the protocol with an amazing degree of automation. Various other coherence protocols [SAR99, SSA01] are also formally verified, mostly using model checking techniques since these scale well for this kind of proof. While these protocols support complex multiprocessor scenarios, verifications is only done at the protocol level, i.e., actual implementations are not considered. The consistency criteria verified are similar to ours, e.g., in case of a read, the result of the last write is returned. Caches are completely abstracted away; the are abstract components that can hold, e.g., valid, dirty, or invalid addresses. The connection of these caches to a physical memory with an additional bus protocol and bursts is not considered. We know of no complete proof of correctness for a cache memory interface at the gate level accessing a memory via a bus protocol, either as a paper proof or using formal methods.

Chapter 4

The VAMP microprocessor

In this chapter, we first describe the VAMP specification, i.e., the programmer's model, and the implementation. We then introduce a key concept in the formal verification of the VAMP, the so-called scheduling functions. With the help of these scheduling functions, we finally give the correctness criteria and the overall correctness proof of the VAMP.

4.1 Programmer's model

The programmer's model presented in this section is based on the work of Daniel Kröning [Krö01].

From a programmer's view, a microprocessor typically consists of some configuration consisting of a memory, a register file, and a program counter. A computation step in this model consists of executing the instruction identified by the program counter; this execution may affect all the components of the configuration, in particular the program counter itself.

For the VAMP configuration, we have three separate register files, a general purpose register file *GPR* with 32 registers, a floating point register file *FPR* which also contains 32 registers, and a special purpose register file *SPR* with 9 registers. We will give more details on the *SPR* registers later on when detailing the interrupt mechanism. The register width is 32 bit. Note that register 0 of the *GPR* always contains 0. For double precision floating point operations, the *FPR* is accessed as a 64 bit wide register file with 16 entries, e.g., the double precision register 0 is an alias for the single precision registers 0 and 1. The VAMP memory M contains 2^{32} bytes.

For implementation reasons discussed in the following section, the VAMP features so-called *delayed PC* with one delay slot. In such a delayed PC architecture, instruction updates to the PC do not effect the next instruction, only the instruction after the next one. Hence, the instruction after a jump instruction is always executed before the actual jump takes place and we have two PCs in the programmer's model, PC' and DPC. In an execution

step, the instruction pointed to by *DPC* is executed, but the PC update of the instruction only affects PC'. Simultaneously, the old value of PC' is written to *DPC* which creates the desired delay slot. Hence, PC' and *DPC* are basically a queue of depth two in the specification.

In summary, a VAMP specification configuration is a 6-tuple (PC', *DPC*, M, *GPR*, *FPR*, *SPR*). We use the notation c_S for a specification configuration in order to distinguish it from the implementation configuration later on. We also introduce the notation c_S^n for the specification configuration *before* the execution of instruction n. Hence, c_S^0 denotes the initial configuration and c_S^7, e.g., denotes the configuration after executing instruction 6, but before executing instruction 7. We also introduce an alternative computation of the programmer's model *without interrupts*, i.e., a specification that just does not react to interrupts. For a computation without interrupts, we use the notation $\tilde{c}_S^n$ in contrast to c_S^n. In the following, we will first give details on the next state computation for $\tilde{c}_S^{n+1}$, then extend it to the full c_S^{n+1}.

We also introduce some functions on specification configurations in order to abbreviate notations. We start with a function *IR* returning the instruction the current configuration is supposed to execute. Note that the VAMP requires instruction fetch to be aligned, i.e., we must have $c_S.DPC \bmod 4 = 0$. Since in case of a violation, we do not want to access the memory at all, we actually have the following equation for the instruction register *IR*.

$$IR(c_S) = \begin{cases} c_S.M[c_S.DPC + 3 : c_S.DPC] & \text{if } c_S.DPC \bmod 4 = 0 \\ 0^{32} & \text{otherwise} \end{cases} \tag{4.1}$$

Note that according to table A.2 in the appendix, 0^{32} actually encodes a `slli` instruction with destination register 0, i.e., a `nop` instruction.

Based on this instruction register *IR*, figure A.1 on page 169 introduces some additional functions like *RD* containing the index of the destination register in a register file. Finally, tables A.1 to A.6 encode the next state of the programmer's model *without interrupts*. We assume that we have a predicate on an instruction word for any instruction according to these tables, e.g., `add`?(*IR*) holds iff both $IR[31:26] = 0^6$ and $IR[5:0] = 10^5$ hold according to table A.2. In addition, we have a predicate `fpu?` for *all* FR-type instructions according to table A.5, and a predicate `mem?` for memory instructions of I- and FI-type from tables A.1 and A.4.

We now formally define a fragment of one computation step in the programmer's model without interrupts. Note that we use primed notation for the next state without interrupts, i.e., c'_S denotes the state one computation step after c_S, and all functions and components on the right-hand side as well as *RD* refer to the unprimed state c_S. Note that memory operations have an effective memory address

$$ea(c_S) := c_S.GPR[RS1(c_S)] + imm(c_S) \tag{4.2}$$

and a width of d bytes as introduced in tables A.1 and A.4. Additionally, memory accesses are required to be aligned, i.e., we demand $ea(c_S) \bmod d = 0$. If this condition is not fulfilled, stores are just ignored and loads have 0 as a result in the programmer's model without interrupts.

$$c'_S.GPR[RD(c_S)] = \begin{cases} c_S.GPR[RS1(c_S)] + imm(c_S) & \text{if } \texttt{addi?}(\texttt{c}_\texttt{S}) \\ c_S.GPR[RS1(c_S)] - imm(c_S) & \text{if } \texttt{subi?}(\texttt{c}_\texttt{S}) \\ c_S.M[ea(c_S)+d-1 : ea(c_S)] & \text{if } \texttt{lw?}(\texttt{c}_\texttt{S}) \wedge \\ & \quad d \,|ea(c_S) \\ 0^{32} & \text{if } \texttt{lw?}(\texttt{c}_\texttt{S}) \wedge \\ & \quad d \not| ea(c_S) \\ \vdots & \vdots \\ c_S.GPR[RD(c_S)] & \text{otherwise} \end{cases} \tag{4.3}$$

The extension of this example to all supported instructions and the full specification configuration is straightforward. Note that we have to take care in order to ensure that register 0 always returns 0. Hence, $c_S.GPR[RS1(c_S)]$ actually denotes 0 in case of $RS1(c_S) = 0$; the same obviously holds for $RS2(c_S) = 0$. In addition, all instructions of the VAMP have the effect

$$\begin{aligned} c'_S.PC' &= c_S.PC' + 4 \\ c'_S.DPC &= c_S.PC' \end{aligned}$$

unless explicitly noted otherwise in the corresponding tables A.1 to A.5. Hence, $c_S.PC'$ is incremented unless we have a branch or jump instruction or an `rfe` and only an `rfe` can access the $c_S.DPC$ directly. Thus, `rfe` is basically a special jump instruction *without delay slot*. Note we will not give a full version of the formal specification in this thesis due to its length. We refer to the semi-formal specification in the tables although and the fully formalized specification in PVS.

We can now recursively define a computation without interrupts with an initial configuration c_S^{init} and the primed notation from above by

$$\begin{aligned} \tilde{c}_S^0 &:= c_S^{init} \\ \tilde{c}_S^{n+1} &:= \tilde{c}_S^{n\prime}. \end{aligned}$$

As a next step, we want to define specification computations with interrupts. We therefore introduce two additional functions on specification configurations, i.e., $CA(c_S)$ and $EData(c_S)$, that return the exception cause and data of the specification configuration, respectively. Table 4.1 summarized the supported exception causes in the VAMP. Note that *reset* plays a special role as an interrupt since it is *external*. The *ill* exception is asserted if $IR(c_S)$ does not decode any instruction according to tables A.1 to A.5.

Index	Name	Type	Maskable	Interrupt
0	*reset*	repeat	no	reset
1	*ill*	repeat	no	illegal instruction
2	*mal*	repeat	no	misaligned memory access
3	*ipf*	repeat	no	page fault on fetch
4	*dpf*	repeat	no	page fault on load/store
5	*trap*	continue	no	trap instruction
6	*ovf*	continue	yes	fixed point overflow
7	*OVF*	continue	yes	floating point overflow
8	*UNF*	continue	yes	floating point underflow
9	*INX*	continue	yes	floating point inexact result
10	*DIVZ*	continue	yes	floating point division by zero
11	*INV*	continue	yes	floating point invalid operation
12	*UNIMP*	continue	no	floating point unimplemented

Table 4.1: Supported interrupts in the VAMP

There can be two reasons for a misaligned memory access, a misaligned instruction fetch $imal(c_S)$ or data memory access $dmal(c_S)$. Therefore, we have the following equation for misalignment:

$$\begin{aligned} CA_S(c_S)[mal] &= imal(c_S) \vee dmal(c_S) && (4.4)\\ imal(c_S) &= c_S.DPC \bmod 4 \neq 0 && (4.5)\\ dmal(c_S) &= ea(c_S) \bmod d \neq 0 && (4.6) \end{aligned}$$

Since we have no address translation in the VAMP, both ipf and dpf are tied to 0. The *trap* exception is raised on the special `trap` instruction. On an arithmetic overflow of an instruction that does not suppress overflows like `addu` does, ovf is signaled. The interrupts 7–11 are raised as requested by the IEEE standard on floating point operations. The *UNIMP* exception, finally, is raised for instructions `fsqrt` and `frem` since these are not supported by the VAMP architecture. Depending on the interrupt, the exception data is computed as follows:

$$EData(c_S) := \begin{cases} c_S.DPC & \text{if } imal(c_S) \vee ipf(c_S) \\ imm(c_S) & \text{if } \texttt{trap}?(c_S) \\ ea(c_S) & \text{if } \texttt{mem}?(c_S) \\ FPUresult(c_S) & \text{if } \texttt{fpu}?(c_S) \\ 0^{32} & \text{otherwise} \end{cases} \qquad (4.7)$$

Note that in the above equation, we used the shorthand *FPUresult* for the result of a floating point instruction. Details on the computation of a

Index	Name	Function
0	*SR*	Status register. Contains interrupts mask bits.
1	*ESR*	Exception status register. Saves *SR* in case of an interrupt.
2	*ECA*	Exception cause register. Saves exception cause in case of an interrupt.
3	*EPC*	Exception *PC*. Saves PC' in case of an interrupt.
4	*EDPC*	Exception *DPC*. Saves *DPC* in case of an interrupt.
5	*EData*	Exception data. Saves additional exception data in case of an interrupt.
6	*RM*	Rounding mode. Encodes currently used rounding mode for all floating point operations.
7	*IEEEf*	IEEE flags register. Required by the IEEE standard to accumulate floating point interrupts.
8	*FCC*	Floating point condition code. Used to store result of floating point comparisons.

Table 4.2: Special purpose registers of the VAMP

$RM[1:0]$	Rounding
00	toward zero
01	to next even
10	toward $+\infty$
11	toward $-\infty$

IEEEf Bit	Floating point interrupt
0	overflow
1	underflow
2	inexact result
3	division by zero
4	invalid operation

Table 4.3: Coding of the registers *RM* and *IEEEf*

result of a floating point instruction according to the IEEE standard are given in [Jac02a]. After introducing CA and $EData$, we have a look at the special purpose register file. The register file in the VAMP contains six registers dealing with interrupts and three as required by the IEEE standard for floating point operations according to table 4.2. Note that the coding for registers RM and $IEEEf$ is given in table 4.3.

There is one detail of the specification without interrupts we have neglected so far since it needs the definition of CA. The IEEE standard [Ins85] requires that every floating-point operation computes 5 exception bits (e.g., overflow and underflow), which are accumulated in status flags. Each flag is set whenever the corresponding exception occurs, and it is reset only through explicit writes to the status flag. In the VAMP, the 5 status flags are stored in the special purpose register $IEEEf$ (cf. table 4.3), which is updated after every FPU instruction, and which can be written and read explicitly by means of moves between the SPR and GPR. Hence, we really have the following relation for special purpose register 7 ($IEEEf$) in computations without interrupts:

$$c'_S.SPR[7] = \begin{cases} c_S.GPR[RS1(c_S)] & \text{if } \texttt{movi2s}?(c_S) \wedge \\ & SA(c_S) = 7 \\ c_S.SPR[7][31:5](c_S.SPR[7][4:0] \vee & \text{otherwise} \\ \qquad CA(c_S)[11:7]) & \end{cases} \tag{4.8}$$

Note that we do not have to explicitly refer to floating point instructions in the above equation since the specification guarantees that these five interrupts can only be generated by floating point instructions anyway. Hence, if we have neither an explicit write by `movi2s` with destination register 7 nor a floating point operation, the above equation just evaluates to an identity.

We now proceed with the integration of interrupts. The status register SR is used to mask certain interrupts according to table 4.1. Note that we support both repeat and continue interrupts, i.e., after handling the interrupt, either the interrupted instruction is executed again or the next instruction after it. In case of multiple interrupts in the same instruction, the one with the smallest index is used in order to decide whether an interrupt is of type repeat. We will not formally define the interrupt level as the smallest cause index as was done in [MP00] for the DLX since this interrupt level is used only for the repeat decision in specification. Note, however, that the interrupt handler software is also supposed to only react to the interrupt with the lowest index, i.e., to compute the interrupt level. For the type of the interrupt, we therefore simply have

$$repeat(c_S) := \bigvee_{i=0}^{i<5} CA(c_S)[i]. \tag{4.9}$$

In order to detect an interrupt, we have to take the interrupt masks form register *SR* into account. We therefore define the masked cause register and base the decision on whether an interrupt occurs on this register.

$$MCA(c_S) := \lambda_{i\in\mathbb{Z}_{32}} \begin{cases} CA(c_S)[i] \wedge c_S.SR[i] & \text{if } i \geq 6 \wedge i \neq 12 \\ CA(c_S)[i] & \text{otherwise} \end{cases} \quad (4.10)$$

$$JISR(c_S) := MCA(c_S) \neq 0^{32} \quad (4.11)$$

Note that for the computation of $repeat(c_S)$, we can simply refer to *CA* instead of *MCA* since $MCA(c_S)[4:0] = CA(c_S)[4:0]$ holds anyway. In case of $JISR(c_S)$, a computation with interrupts calls an interrupt service routine starting at a fixed address *SISR* and saves some additional values which we will define later on. Note, however, that without $JISR(c_S)$, computation steps with interrupts are equal to those without, i.e., we have $\neg JISR(c_S^n) \implies c_S^{n+1} = c_S^{n\prime}$. Therefore, the following proposition trivially holds.

Proposition 4.1.1 *As long as no interrupt occurs, both specification computations are equal, i.e.,* $\forall n \in \mathbb{N}$*:*

$$(\forall m \in \mathbb{Z}_n : \neg JISR(\tilde{c}_S^m)) \implies \tilde{c}_S^n = c_S^n$$

Note also that you can exchange $\neg JISR(\tilde{c}_S^m)$ in the left-hand side of the above proposition by $\neg JISR(c_S^m)$ while preserving the correctness of the right-hand side. Therefore, we only have to give a definition of c_S^{n+1} for the $JISR^n$ case. Note also that we refer to $c_S^{n\prime}$ in this definition, i.e., the next state after c_S^n in a computation without interrupts. For the all components apart form *SPR*, the definition is straightforward:

$$\begin{aligned} c_S.DPC^{n+1} &= SISR \\ c_S.PC'^{n+1} &= SISR + 4 \\ c_S.M^{n+1} &= repeat(c_S^n)?\, c_S^n.M : c_S^{n\prime}.M \\ c_S.GPR_{n+1} &= repeat(c_S^n)?\, c_S^n.GPR : c_S^{n\prime}.GPR \\ c_S FPR^{n+1} &= repeat(c_S^n)?\, c_S^n.FPR : c_S^{n\prime}.FPR \end{aligned}$$

In the *SPR*, we save the masked exception cause and data as well as the PCs of the instruction we are supposed to return to after handling the interrupt. All interrupts are masked when entering the interrupt service routine. Note that when saving the interrupt masks, we also have to save the mask that the instruction after the return from the interrupt handler is supposed to see, i.e., in case of a continue interrupt, we have to save the

primed version of *SR*.

$$c_S.SPR^{n+1} = \lambda_{i\in\mathbb{Z}_9}\begin{cases} 0^{32} & i = SR \\ repeat(c_S^n)?\, c_S^n.SPR[0] : c_S^{n\prime}.SPR[0] & i = ESR \\ MCA(c_S^n) & i = ECA \\ repeat(c_S^n)?\, c_S^n.PC' : c_S^{n\prime}.PC' & i = EPC \\ repeat(c_S^n)?\, c_S^n.DPC : c_S^{n\prime}.DPC & i = EDPC \\ EData(c_S^n) & i = EData \\ repeat(c_S^n)?\, c_S^n.SPR[i] : c_S^{n\prime}.SPR[i] & \text{otherwise} \end{cases} \tag{4.12}$$

This completes the formal definition of the programmer's model. When proving the VAMP implementation correct later on, we will show that it simulates the above programmer's model.

4.2 Implementation

Implementation and formal verification of the floating point units of the VAMP are given by Jacobi [Jac02a]. Apart from the Tomasulo correctness proof, Kröning [Krö99,Krö01] also developed an initial unverified VAMP implementation without memory unit which served as a starting point for our work. Note, however, that we had to fix numerous bugs in this implementation before we were able to prove its correctness.

A typical pipelined in-order microprocessor splits instruction execution into four phases, namely instruction fetch, decode, execute, and writeback. During instruction fetch, the instruction is loaded from the memory; decode reads the source operands of the instruction from the register file. The execute phase computes the result of an instruction which is written to the register file during writeback. As a consequence of this classical pipeline structure, instruction decode has to be stalled until all source operands of an instruction are available. As long as the execute phase has a constant small latency, overall performance is hardly affected by this stalling. However, the execute phase typically has a very wide range of possible latencies, from simple single-cycle operations to memory accesses with cache misses that may take hundreds of cycles. In this case, performance may notably increase by introducing out-of order execution, i.e., by allowing an instruction to overtake a previous instruction that still waits for operands. One of the best known out-of order execution schemes is the so-called Tomasulo [Tom67] algorithm that was invented and patented in 1967.

4.2.1 Tomasulo algorithm

The basic Tomasulo algorithm also allows instructions to leave the pipeline out-of order. However, most microprocessor support interrupts, i.e., as a

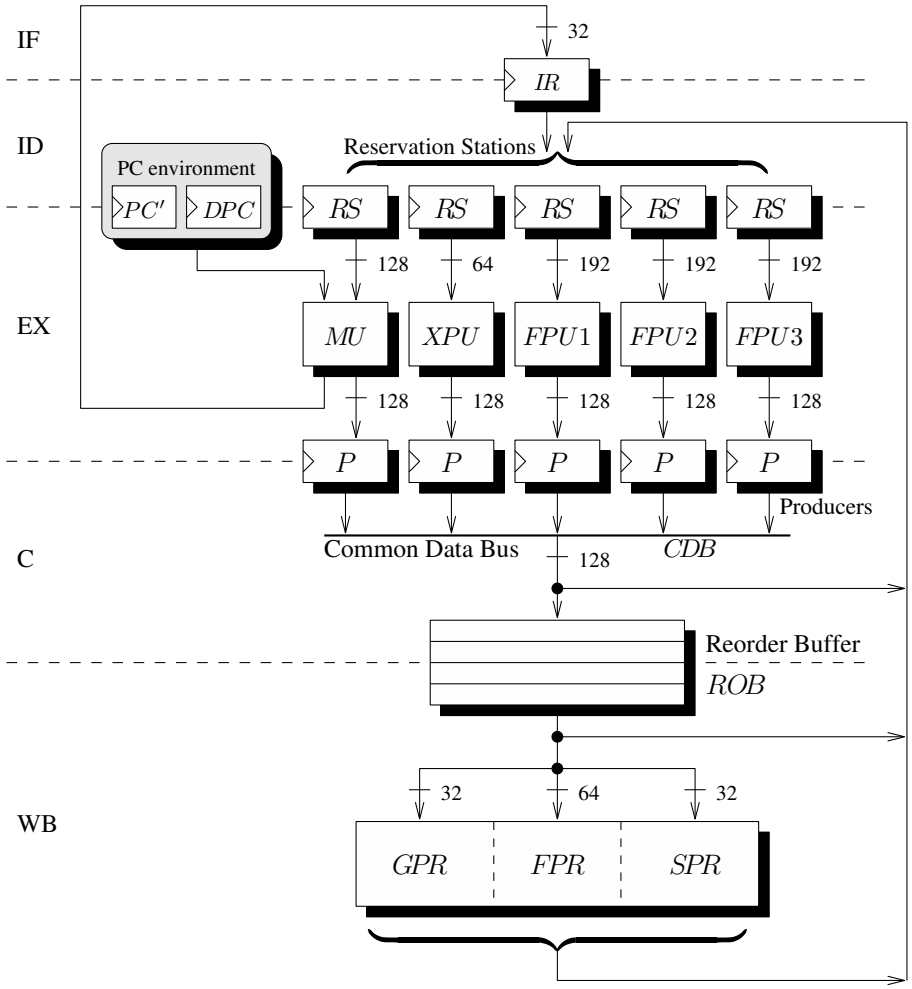

Figure 4.1: The VAMP data paths

reaction to some events, sequential program execution may be interrupted and a so-called handler routine is called. At some later cycle, this handler may terminate and sequential execution of the interrupted code continues with the next instruction. In order to implement this behaviour, interrupts have to be precise, i.e., if some instruction is interrupted, then all previous instructions are already executed, but no later instruction has been executed. Hence, for a scenario with precise interrupts, the Tomasulo algorithm is usually enhanced by a so-called *reorder buffer* that ensures that instructions leave the pipeline in order, although they may have been executed out-of order inside the pipeline. In the following, we therefore describe the Tomasulo algorithm with reorder buffer as implemented in the VAMP according to figure 4.1.

In order to implement out-of order execution, a so-called *producer table*

is added to the register file. This producer table contains for each register a valid bit and a tag. An active valid bit indicates that no instruction currently in the pipeline writes to the corresponding register, i.e., the register file contains the "correct" data an instruction currently being decoded may want to read. In case of an invalid register, there is some instruction in the pipeline that writes this register; this instruction is identified by the tag from the producer table.

Instruction decode is split into *two* phases, namely issue and dispatch. Issue is integrated into the "old" instruction decode and thus still occurs in order. During issue, all operands that are available are read; for the missing ones, the corresponding tag from the register file is inserted such that later on, the actual data belonging to this tag may be added. The instruction with all its source operands—either as actual data or mere tags—is passed to a new data structure called *reservation station*, a new tag is reserved for this instruction, the destination register of this instruction is marked as invalid and the tag set to the instruction's tag, and a new entry is added to the reorder buffer which is basically just a Fifo queue.

In a reservation station, instructions wait until all their operands are available. This is accomplished by *bus snooping*, i.e., all reservation stations snoop on the *common data bus* for data that is paired with the tag they are expecting data from. The actual sources of data and tag on the common data bus will be introduced later.

As soon as an instruction in a reservation stations has all its source operands available, instruction dispatch may occur out-of order. During dispatch, an instruction leaves a reservation station and enters an execution unit with its corresponding tag. Instruction dispatch is integrated into the first cycle of instruction execution. Note that there can be an arbitrary number of reservation stations per execution unit; even a dynamic allocation from reservation stations to execution units is possible. Execution units may arbitrarily reorder instructions internally; therefore, the instruction's tag is used to identify the results.

If an instruction is about to leave an execution unit, its result and tag are stored in a *producer* register. Therefore, there is exactly one producer register for each execution unit. The producer registers of all instructions are connected to the common data bus introduced before. An arbiter on the common data bus decides which instruction may *complete*, i.e., write its result from a producer register into the reorder buffer and mark the corresponding result in the reorder buffer as valid. In addition, this instruction's result becomes visible on the common data bus and may be snooped by reservation stations.

As soon as the oldest result in the reorder buffer becomes valid, writeback may occur and the result from the reorder buffer is finally written back to the register file. Note that writeback thus occurs in-order since entries are added to the *ROB* in-order during instruction issue. If the instruction's tag from

the reorder buffer matches the tag in the producer table of the destination register, no further instruction in the pipeline writes the register and it may be marked as valid. Otherwise, it has to be left invalid.

After introducing all the components, we are able to give more details for data forwarding during decode. As a first step, the register file is considered. If it does not hold valid data, the reorder buffer entry of the producing instruction—identified via its tag—is considered. If the reorder buffer does not hold a valid result either, the common data bus is snooped. Only if there is also no valid data here, the tag of the producing instruction is issued into the reservation station since the producing instruction is in a execution unit which it will eventually leave. It will then put its result on the common data bus and it may be snooped by the reservation station.

4.2.2 VAMP implementation

In the VAMP, we have five separate execution units: a fixed point unit XPU, a memory unit MU, and thee specialized floating point units $FPU1$, $FPU2$, and $FPU3$ for addition/ subtraction, multiplication/ division, and conversion/ testing, namely. Design and correctness proofs of these three FPUs can be found in [Jac02a]. The fixed point unit is just an ALU with a shifter; its correctness follows immediately from the correctness of basic circuits since it is purely combinational. For the memory unit that performs both memory operations and instruction fetch, we will give an overview of the implementation and correctness proof in section 4.4.2.

There are four reservation stations for the fixed point unit; all the other execution units have only one reservation station each. Hence, there is a total of eight reservation stations and we also have eight reorder buffer entries. This makes for a tag width of 3 bit. Special instructions that do not really compute anything, e.g., jump or data transfer instructions, may bypass the execution units entirely; they are issued directly into the reorder buffer. Note that these instructions have to be stalled in decode until all their operands are available since they cannot snoop for operands in the reorder buffer.

An instruction in the VAMP has up to *six* 32-bit source operands, namely four containing two 64-bit operands for double precision operations, the rounding mode, and the status register as required by the IEEE 754 [Ins85] standard. At first glance, an instruction in the VAMP produces two results for double precision floating point operations. However, as we introduced in the programmer's model, interrupts handling depends on CA and $EData$ which are basically also results of any instruction. While these results are discarded as long as no interrupt occurs, they are really saved in case of an interrupt. Hence, we consider a Tomasulo instantiation with *four* results per instruction by adding CA and $EData$. These two additional results will be crucial in proving correctness with interrupts in section 4.5.

Since instruction fetch is not part of the Tomasulo algorithm, we have

to add some fetch mechanism in the VAMP implementation. In the programmer's model, we introduced the delayed PC scheme with a queue of two PCs. We will now argue on the implementation why this approach was taken. We have separate instruction fetch and decode stages in the VAMP as depicted in figure 4.1. While some branch instruction I_i is in the decode stage, we already fetch the next instruction I_{i+1}. In order to evaluate the branch condition prior to the next instruction fetch, one would basically have to do instruction decode and fetch in one cycle which increases cycle time considerably. Therefore, only two feasible solutions remain:

1. Predict the program counter of the next instruction and perform a rollback in case of misprediction. If we have speculated correctly, instruction throughput does not suffer. On a misprediction, however, we have to pay a penalty be performing a rollback and fetching the correct instruction. On the other hand, prediction does not influence the sequential programmer's model at all.

2. Change the semantics of the assembler instruction set such that jumps and branches take effect only with a delay of one instruction. Hence, after any branch or jump, the next instruction is *always* executed before the actuals branch or jump is taken. This instruction after a branch or jump is called a delay slot. By reordering instructions while preserving program semantics in an assembler, about 80% of the delay slots can typically be filled by meaningful instruction.

We decided to use the delayed branch mechanism in the VAMP in analogy to the DLX from [MP00] since all our assembly code for the DLX was based on the delayed branch semantics anyway and we wanted to keep upwards compatibility as far as possible. We implement the delayed branch with the equivalent delayed PC mechanism [MP00] where *all* PC computations are delayed by one instruction. Hence the queue of depth two for the PC in the programmer's model and, correspondingly, two PC registers DPC and PC' in the implementation. Note that the computation of the next DPC and PC' in the implementation, respectively, corresponds to the specification—the source operands of the specification are only replaced with the corresponding forwarded source operands from the implementation whose correctness is guaranteed by the Tomasulo algorithm. Note that instruction fetch in the implementation does not simply occur with DPC as in the specification. Details in section 4.4.3 on the correctness of instruction fetch.

The result of instruction fetch is saved in the $S1$ registers, i.e., the instruction registers IR, and two flags for the possible exceptions during fetch, i.e., $imal_I$ and ipf_I for misaligned instructions and a page-fault on fetch, respectively. From this $S1$ register, instruction issue as introduced in the Tomasulo algorithm may occur. Note that we introduced a function IR on the programmer's model that returned the fetched instruction. In our cor-

rectness criteria in section 4.3.2, we will therefore map register $S1.IR$ to IR and $S1.imal_I$ to $imal_S$.

As we have already seen in the programmer's model in equation (4.8), the *IEEEf* special purpose register is kind of unique because it is updated by many instructions. Since new exception bits have to be or-ed to the *IEEEf* register for every FPU instruction, *IEEEf* is formally both source and destination operand of every floating point instruction. If *IEEEf* would be handled by the Tomasulo scheduler as a regular register, the number of source operands would increase to seven and destination registers to five. What is worse, at most one floating point instruction could be in all three FPUs *together* at any time.

In order to significantly increase performance and keep the size of reservation stations, *CDB*, and *ROB* small, we instead update *IEEEf* during writeback in a special way: Register *IEEEf* is neither Tomasulo source nor destination operand of any floating point instruction; only for a special move `movs2i` with source *IEEEf* or a special move `movi2s` with destination *IEEEf* have register *IEEEf* as Tomasulo source or destination operand, respectively. Note that these special moves are trivially disjoint from floating point instructions. The update of *IEEEf* for any floating point instruction is performed differently. The exception cause *CA* of an instruction is part of the result in the *ROB* anyway. When an instruction that does *not* have *IEEEf* as Tomasulo destination register, i.e., any instruction other than `movi2s` with destination register *IEEEf*, is written back from the *ROB* into the register file, the respective bits of its result *CA* are logically or-ed to the current *IEEEf* register just like in the programmer's model.

However, the forwarding mechanism of the standard Tomasulo algorithm is then no longer correct for explicit reads of *IEEEf*, i.e, a `movs2i` with source register *IEEEf* requires additional consideration. We therefore add a hardware synchronization for any instruction I_i explicitly reading *IEEEf*: instruction issue of I_i is stalled until the reorder buffer has run empty, i.e., until no other instruction that might possibly update the *IEEEf* register via the 'new' mechanism for floating point instructions is alive in the VAMP processor. We will prove this extension of the Tomasulo algorithm correct in section 4.4.1.

Since interrupts are not part of the Tomasulo algorithm, we have to extend the Tomasulo implementation. Because we already have $CA(c_I)$ and $EData(c_I)$ as part of the result in the reorder buffer, we can compute $repeat(c_I)$, $MCA(c_I)$, and $JISR(c_I)$ during writeback just like the corresponding functions in the programmer's model, i.e., equations (4.9) to (4.11). As long as $JISR(c_I)$ does not hold, the VAMP is a standard Tomasulo implementation; otherwise, the VAMP is flushed, i.e., all reservation stations, execution units, producer, the reorder buffer, and the instruction register are cleared and all registers in the register file are marked as valid. In addition, actions corresponding to those defined for the programmer's model

with interrupts are taken according to equation (4.12), e.g., the PCs are set to the start of the interrupt service routine and the status register is cleared. Since we have to save the PCs of the interrupted instruction or the ones of the instruction thereafter, we have to extend the reorder buffer by some additional PC registers for each entry. During instruction issue, both the current as well as the next value of PC' and DPC are written to the reorder buffer. In case of an interrupt, two of these values depending on $repeat_I$ are stored in the corresponding special purpose registers.

An implementation configuration is a 17-tuple (PC', DPC, M, GPR, FPR, SPR, $S1$, RS, P, ROB, $ROBhead$, $ROBtail$, $ROBcount$, MU, $FPU1$, $FPU2$, $FPU3$). As introduced by Kröning [Krö01], $ROBhead$ and $ROBtail$ point to the head and tail of the reorder buffer, respectively, while $ROBcount$ returns the number of entries currently in the ROB. This additional counter is needed since in case of $ROBhead = ROBtail$, the ROB can be either full or empty. Note that we do not have a component part for the fixed point unit XPU since it is purely combinational. Similar to the specification configuration, we denote implementation configurations with c_I. Note that it is no coincidence that the memory configuration is denoted by $c_I.M$ similar to a correct memory interface M_I according to definition 1.5.2—the VAMP implementation memory just *is* a correct memory interface.

Note, however, that in the definition 1.5.2, we require a *clear* in the initial cycle and only define the memory content *after* the initial cycle. In other words, we assume an arbitrary initial state and an initial clear and thus get some valid initial state in cycle 1. However, for the following arguments, we will argue about valid initial states in cycle 0 since this facilitates the proof; only at the very end of the overall proof, we will show that given an arbitrary configuration and a clear in an initial cycle, we obtain a valid initial configuration in the next cycle and can apply the correctness proof to this valid initial configuration. The clean solution to this problem would involve redoing the proofs in chapter 2 and 3 to also assume a valid initial configuration instead of an arbitrary one and an initial clear. However, since this would involve considerable effort, we will not do so. Instead, we introduce an intermediate definition for the memory content of the memory interface that assumes a valid initial configuration and allows for write accesses in this initial configuration. We therefore define the memory content $c_I.M$ by

$$\begin{aligned} c_I^0.M &:= init\_mem \\ \left|c_I^{t+1}.M[\langle ad\rangle]\right|_b &:= \begin{cases} \left|din^t\right|_b & \text{if } M_I.bw(ad,b)^t \\ \left|c_I^t.M[\langle ad\rangle]\right|_b & \text{otherwise} \end{cases} \end{aligned}$$

and note that if the input and output sequence of M_I are given by inp and out and the corresponding sequences of $c_I.M$ are given by $inp' := \lambda_{t\in\mathbb{N}} inp(t+1)$ and $out' := \lambda_{t\in\mathbb{N}} out(t+1)$, respectively, we trivially have $M_I^{t+1} = c_I^t.M$ for any $t \in \mathbb{N}$. It is also obvious that we can apply decomposition to $c_I.M$ as

desired, i.e., $c_I^{t+k}.M = c_I[c_I^t.M]^k.M$. We will base our further arguments on $c_I.M$ and close the gap to the proved M_I in section 4.6.

In addition c_I^t denotes the implementation configuration at the beginning of hardware cycle t. Similar to the second specification $\tilde{c}_S$, we introduce an alternative implementation $\tilde{c}_I$ that does not react to interrupts. In analogy to the specification configurations, the following proposition trivially holds.

Proposition 4.2.1 *As long as no interrupt occurs, both variants of the implementation computations are equal, i.e.,* $\forall t \in \mathbb{N}$*:*

$$(\forall t' \in \mathbb{Z}_t : \neg JISR(\tilde{c}_I^{t'})) \implies \tilde{c}_I^t = c_I^t$$

Up to now, we did not give any details on initial configurations. Therefore, we now introduce a predicate capturing that an implementation configuration is initial, i.e., empty, and a function mapping an implementation configuration to a specification configuration. Later on, we then assume an arbitrary, but fixed initial implementation configuration. The initial specification configuration is then derived from the implementation configuration by means of the mapping function.

Definition 4.2.2 *We call an implementation configuration* c_I initial, *i.e.,* $init?(c_I)$*, iff it represents an empty VAMP with valid registers, i.e., all reservation stations, execution units, and producer registers, the ROB, and the decode stage is empty and all registers are valid. Formally, we have:*

$$\begin{aligned} init?(c_I) \quad :\Longleftrightarrow \quad & \neg c_I.S1.full \wedge c_I.ROBcount = 0 \wedge \\ & empty?(c_I.MU) \wedge empty?(c_I.FPU1) \wedge \\ & empty?(c_I.FPU2) \wedge empty?(c_I.FPU3) \wedge \\ & c_I.DPC = SISR \wedge c_I.PC' = SISR + 4 \wedge \\ & (\forall x \in \mathbb{Z}_8 : \neg c_I.RS[x].full) \wedge (\forall x \in \mathbb{Z}_5 : \neg c_I.P[x].full) \wedge \\ & (\forall x \in \mathbb{Z}_{32} : c_I.GPR[x].valid \wedge c_I.FPR[x].valid) \wedge \\ & (\forall x \in \mathbb{Z}_9 : c_I.SPR[x].valid) \end{aligned}$$

Note that we do not need $empty?(c_I.XPU)$ *since the XPU is purely combinational. Additionally, we will not give any details on the predicate* $empty?$ *since this would involve the complete pipeline structure of the three floating point units.*

We define a function $spec\_conf$ *that creates a specification configuration from an initial implementation configuration. This function just takes all the relevant parts from the implementation, i.e., we have the 6-tuple*

$$\begin{aligned}
spec_conf(c_I).PC' &= c_I.PC' \\
spec_conf(c_I).DPC &= c_I.DPC \\
spec_conf(c_I).M &= c_I.M \\
spec_conf(c_I).GPR &= \lambda_{x\in\mathbb{Z}_{32}} c_I.GPR[x].data \\
spec_conf(c_I).FPR &= \lambda_{x\in\mathbb{Z}_{32}} c_I.FPR[x].data \\
spec_conf(c_I).SPR &= \lambda_{x\in\mathbb{Z}_{9}} c_I.SPR[x].data
\end{aligned}$$

As mentioned above, we so far assumed an arbitrary, but fixed initial implementation configuration. Since we have to argue explicitly about several special initial configurations in the following sections, we employ the suffix-notation $[c_{init}]$ as introduced in definition 1.2.7 on page 6. Hence, if $init?(c_I^t)$ holds, $\tilde{c}_I[c_I^t]^k$ denotes a state after k computation steps without interrupts from the initial state c_I^t which was reached by t steps with interrupts from the arbitrary, but fixed initial configuration.

We also want to use the above notation for components of the specification. This is achieved by means of the $spec\_conf$ function introduced above in definition 4.2.2. Hence, $c_S[c_I^t]^i$ denotes the specification configuration after i steps with interrupts starting from the initial configuration $spec\_conf(c_I^t)$.

Proposition 4.2.3 *Any computation with $t+k$ steps is equivalent to a computation with k steps starting from the configuration after t steps as initial configuration. Formally, we have $\forall t, k \in \mathbb{N}$:*

$$init?(c_I^t) \implies c_I^{t+k} = c_I[c_I^t]^k$$

We once again omit the proof due to its triviality. Note that the corresponding decomposition property for c_S^{t+k} does *not* generally hold since we derive the initial configuration for the specification from the implementation. When it is clear from the context that we are referring to the implementation, we will omit the prefix c_I in the following sections.

4.3 Correctness criteria

4.3.1 Scheduling functions

When trying to prove a microprocessor correct, one has to argue about an instruction that is currently in some pipeline stage, i.e., the instruction in decode or in some reservations station. We therefore introduce the concept of scheduling functions [MP00,BJK+03]. A scheduling function maps a pipeline stage in some cycle to the index of the instruction according to the sequential programmer's model. If k is some stage in the CPU and t is a cycle, $sI(k,t)$ returns the instruction that is in stage k in cycle t as a natural number.

Hence, scheduling functions are some kind of infinite tags. Since they are only used in the proofs of the design, however, the actual hardware remains fully synthesizable.

We distinguish two kinds of scheduling functions, those for the visible registers, i.e., registers that are also part of the programmer's model, and the invisible registers. First of all, let us consider scheduling functions for invisible registers without interrupts, i.e., scheduling functions based on $\tilde{c}_I^t$ computations. Definition of these scheduling functions is straightforward: If an instruction in stage k in cycle t proceeds to stage k', denoted by $ue.k' \rightarrow k$, we set $sI(k', t+1) := sI(k,t)$ while otherwise, the value of the scheduling function remains unchanged. As initial value, we choose -1.

$$\begin{aligned} sI(k,0) &:= -1 \\ sI(k,t+1) &:= \begin{cases} sI(k',t) & \text{if } ue.k' \rightarrow k \\ sI(k,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.13}$$

This covers all cases apart from the decode stage. There, we need an additional definition: The scheduling function for the decode stage, i.e., the $S1$ registers, is incremented iff a new instruction is fetched; otherwise, it remains unchanged. This reflects the fact that instruction decode occurs in order. Hence, the decode scheduling function returns -1 before the first instruction has entered the instruction register; otherwise, it returns the index of the instruction in the instruction register.

In order to formalize this definition for the decode stage, we first introduce the update enable signals $ue.0$ and $ue.1$ for instruction fetch and decode, respectively. Signal $ue.0$ denotes that an instruction is currently being fetched into the instruction register; $ue.1$ denotes that the instruction in the instruction register is being issued into some reservations station or directly into the reorder buffer. Note that according to the Tomasulo algorithm, instruction issue has to be stalled on two conditions, namely if the reorder buffer is full and no writeback occurs or, secondly, if we want to issue into some reservation station, but neither an empty reservation station or one just becoming empty by dispatching is available. These conditions are formalized in [Krö01]; we denote a signal computing these conditions by tom_issue_stall.

In addition, we have to stall instructions directly issued into the reorder buffer until their operands become available. Issuing directly into the reorder buffer occurs on an instruction fetch with either misalignment or a page fault; additionally, illegal instructions, `trap` instructions, as well as `j` or `jal` instructions are issued into the reorder buffer *without* needing to read any operands. In contrast, the remaining branch and jump instructions and `movi2s` all read $GPR[RS1]$, `movs2i` reads $SPR[SA]$, and `rfe` reads $SPR[ESR]$ in order to copy it into $SPR[SR]$. These instructions are all issued into the reorder buffer while actually needing to reed some operand; they are summa-

rized by the signal $issue\_R$ as introduced in the following equation (4.14).

In addition, we denote by $source\_op[0].valid$ that the first source operand obtained by Tomasulo forwarding holds valid data; once again, for a detailed definition for $source\_op$, we refer to [Krö01]. In summary, we have to stall instruction issue if $issue\_R$ holds, but $source\_op[0].valid$ does not. As we have already mentioned, our special implementation of *IEEEf* forces us to stall any `movs2i` that reads *IEEEf* until the reorder buffer is empty. Finally, `rfe` also reads $SPR[EPC]$ and $SPR[EDPC]$ in order to copy them to PC' and DPC, respectively. We therefore stall `rfe` until these two source register are valid according to their producer table, i.e., we do not employ the Tomasulo forwarding algorithm in order to obtain these operands. The overall stall signals of stage decode, $stall.1$ is summarized in equation (4.15).

$$\begin{aligned} issue\_R &= (\texttt{beqz}? \vee \texttt{bnez}? \vee \texttt{jr}? \vee \texttt{jalr}? \vee \\ &\quad \texttt{rfe}? \vee \texttt{movi2s}? \vee \texttt{movs2i}?)(S1.IR) \end{aligned} \tag{4.14}$$

$$\begin{aligned} stall.1 &= S1.full \wedge (tom\_issue\_stall \vee \\ &\quad issue\_R \wedge \neg source\_op[0].valid \vee \\ &\quad \texttt{rfe}?(S1.IR) \wedge \neg(SPR[EPC].valid \wedge \\ &\qquad SPR[EDPC].valid) \vee \\ &\quad \texttt{movs2i}?(S1.IR) \wedge SA(S1.IR) = IEEEf \wedge \\ &\qquad \neg eq_4(ROBcount, 0^4)) \end{aligned} \tag{4.15}$$

$$ue.1 = S1.full \wedge \neg stall.1 \tag{4.16}$$

$$ue.0 = \neg ibusy \wedge \neg stall.1 \tag{4.17}$$

$$\begin{aligned} S1.full^0 &= 0 \\ S1.full^{t+1} &= ue.0^t \vee stall.1^t \end{aligned} \tag{4.18}$$

With the above definitions, the decode scheduling function is easily derived by just incrementing it on $ue.0$.

$$\begin{aligned} sI(dec, 0) &:= -1 \\ sI(dec, t+1) &:= \begin{cases} sI(dec,t)+1 & \text{if } ue.0^t \\ sI(dec,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.19}$$

Note that the scheduling functions as introduced above always return the index of an instruction currently in some invisible register; you might therefore call them *register-based*. For visible registers, we introduce different scheduling function. If a visible register R is updated at the end of some stage k as indicated by a signal $ue.k^t$ in analogy to $ue.0^t$ and $ue.1^t$, we have

the following definition for the scheduling function of this visible register R:

$$\begin{aligned} sI(R,0) &:= 0 \\ sI(R,t+1) &:= \begin{cases} sI(k,t)+1 & \text{if } ue.k^t \\ sI(R,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.20}$$

This definition is straightforward since $sI(k,t)$ is the instruction in stage k and if $ue.k^t$ holds, $sI(k,t)$ updates the visible register and therefore, in cycle $t+1$, register R contains the value that the *next* instruction, i.e., $sI(k,t)+1$, is supposed to read. Note that this is a generalization of the scheduling functions for visible registers introduced in [Krö01]. There, the register file is the only visible register; its scheduling function $sI(wb,t)$ is incremented if a writeback occurs according to equation (4.21). However, Kröning actually proves that in case of a writeback, the scheduling function $sI(wb,t)$ equals the one of the implementation register at the head of the reorder buffer; hence, $sI(wb,t)$ could alternatively be defined with out visible register scheduling function according to equation (4.20).

$$\begin{aligned} sI(wb,0) &= 0 \\ sI(wb,t+1) &= \begin{cases} sI(wb,t)+1 & \text{if } writeback^t \\ sI(wb,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.21}$$

For the PCs as visible registers, we choose the issue scheduling function from Kröning [Krö01] as given by equation (4.22) and prove later on as a part of lemma 4.3.1 that its definition according to Kröning equals our notion of a visible register scheduling function.

In addition, we introduce a scheduling function for instruction fetch that return the index of the *next* instruction to be fetched into the pipeline. We will only use this scheduling function in order to claim correctness of the PC we actually use during instruction fetch. This scheduling function is initialized with 0 for $t = 0$ and incremented on $ue.0$ just like the decode scheduling function. Therefore, we can trivially conclude $sI(fetch,t) = sI(dec,t)+1$. In the following, we examine how the decode scheduling function relates to instruction fetch and issue.

According to the definition of Kröning, the issue scheduling function is incremented on $ue.1^t$. As a part of the following proof, we will actually verify that this scheduling function equals the general visible register scheduling function for the PCs as introduced in this thesis.

$$\begin{aligned} sI(issue,0) &:= 0 \\ sI(issue,t+1) &:= \begin{cases} sI(issue,t)+1 & \text{if } ue.1^t \\ sI(issue,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.22}$$

Lemma 4.3.1 *At any cycle t, the following relation holds for the scheduling functions for fetch, decode, and issue:*

$$sI(issue,t) = \begin{cases} sI(dec,t) & \textit{if } S1.full^t \\ sI(dec,t)+1 & \textit{otherwise} \end{cases}$$

$$sI(issue,t) = \begin{cases} sI(fetch,t)-1 & \textit{if } S1.full^t \\ sI(fetch,t) & \textit{otherwise} \end{cases}$$

Proof: Note that since $sI(fetch,t) = sI(dec,t)+1$ holds, the second equation immediately follows from the first; therefore, it is sufficient to show the first equation by induction.
Induction base ($\mathbf{t=0}$): Initially, we have $sI(issue,t) = 0 = sI(dec,t)+1$ and $\neg S1.full^0$ which finished the induction base.
Induction step ($\mathbf{t \to t+1}$): We have to show

$$sI(issue,t+1) = \begin{cases} sI(dec,t+1) & \text{if } S1.full^{t+1} \\ sI(dec,t+1)+1 & \text{otherwise} \end{cases}$$

We split cases on instruction issue and instruction fetch in cycle t, i.e., on the valued of $ue.0^t$ and $ue.1^t$.

1. Let neither instruction fetch nor issue occur in cycle t, i.e., $\neg ue.0^t \wedge \neg ue.1.t$. We then have $sI(issue,t+1) = sI(issue,t)$, $sI(dec,t+1) = sI(dec,t)$, and $S1.full^{t+1} = S1.full^t$. The claim is then concluded with the induction hypothesis.

2. Let both instruction fetch and issue occur in cycle t, i.e, $ue.0^t \wedge ue.1^t$. We then have both $S1.full^{t+1}$ and $S1.full^t$ and we can apply the induction hypothesis in order to conclude

   $$sI(issue,t+1) = sI(issue,t)+1 = sI(dec,t)+1 = sI(dec,t+1)$$

3. Let instruction issue occur, but no fetch in cycle t, i.e., $ue.1^t \wedge \neg ue.0^t$. This leads to $\neg S1.full^{t+1}$ and $S1.full^t$; we conclude with the induction hypothesis that

   $$sI(issue,t+1) = sI(issue,t)+1 = sI(dec,t)+1 = sI(dec,t+1)+1$$

4. Let finally instruction fetch without issue occur in cycle t which means $ue.0^t \wedge \neg ue.1^t$. This leads to $S1.full^{t+1}$ and $\neg S1.full^t$; with the induction hypothesis, we have

   $$sI(issue,t+1) = sI(issue,t) = sI(dec,t)+1 = sI(dec,t+1)$$

   and the proof is finished. □

Finally, we have to introduce the scheduling function for the last visible register, the memory. In the VAMP, we have a memory stage mem inside the memory unit MU with its corresponding scheduling function $sI(mem, t)$. This obviously is a scheduling function for invisible registers. In the manner introduced by equation (4.20), we can derive a scheduling function for the visible memory from it. Since not all instructions enter the memory unit, the approach of Kröning that only increments the visible register scheduling function does not work for our visible memory. Hence, we had to introduce the more general visible register scheduling function. The updated enable of the visible memory is given if an actual memory access finishes in the current cycle, i.e., misaligned access are ignored for the scheduling function. We therefore have

$$sI(M_I, t+1) := \begin{cases} sI(mem, t) + 1 & \text{if } (mw_I^t \vee mr_I^t) \wedge \neg \tilde{c}_I^t.M.dbusy \\ sI(M_I, t) & \text{otherwise} \end{cases} \tag{4.23}$$

In the VAMP, there is only one reservation station for the memory unit. It is easy to prove that this is sufficient to guarantee in order memory access, i.e, $sI(mem, t)$ is increasing on t. Since no instruction can enter the memory unit twice by the correctness of the Tomasulo scheduler, we also know $i \geq sI(mem, t) + 1$ if instruction i enters the memory unit in cycle t by dispatch from the memory RS. Hence, $sI(M_I, t)$ is also increasing on $t \in \mathbb{N}$.

In order to claim correctness with interrupts, however, we also need the definition of some scheduling function on c_I^t. Therefore, we introduce one additional scheduling function based on the actual c_I that fully integrates interrupts, i.e. $sI(inst, t)$ is incremented if either a writeback or an interrupt occurs; it is initialized with 0. In terms of the above definition of scheduling functions for visible registers, the update enable for the visible register files with interrupts comprises both normal writeback and interrupts.

$$\begin{aligned} sI(inst, 0) &= 0 \\ sI(inst, t+1) &= \begin{cases} sI(inst, t) + 1 & \text{if } writeback(c_I^t) \vee JISR(c_I^t) \\ sI(inst, t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.24}$$

Proposition 4.3.2 *Until one cycle* after *the first interrupt, the scheduling function counting instructions and interrupts and the writeback scheduling function are equal, i.e.,* $\forall t \in \mathbb{N}$*:*

$$(\forall k \in \mathbb{Z}_{t-1} : \neg JISR(c_I^k)) \implies sI(inst, t) = sI(wb, t)$$

Proof: We show the claim by induction on t.
Induction base ($\mathbf{t = 0}$): Initially, we have $c_I^0 = \tilde{c}_I^0$ and $sI(inst, 0) = 0 = sI(wb, 0)$ and the induction base is finished.
Induction step ($\mathbf{t \to t + 1}$): Let $\forall k \in \mathbb{Z}_t : \neg JISR(c_I^k)$ hold. According to proposition 4.2.1, we can conclude $c_I^t = \tilde{c}_I^t$. We split cases on $JISR(c_I^t)$.

1. Let $JISR(c_I^t)$ hold. Since $c_I^t = \tilde{c}_I^t$ holds and $JISR(c_I^t)$ is only raised if the implementation without interrupts performs a writeback, we conclude that $\tilde{c}_I^t$ performs a writeback. Hence, we have $sI(inst, t+1) = sI(inst, t) + 1$ and $sI(wb, t+1) = sI(wb, t) + 1$. We finish the case with the induction hypothesis.

2. Let $\neg JISR(c_I^t)$ hold. Both $sI(inst, t+1)$ and $sI(wb, t+1)$ are then incremented iff a writeback occurs. Since the induction hypothesis guarantees $sI(inst, t) = sI(wb, t)$, the proof is finished. □

4.3.2 Correctness Invariant

In this section, we introduce correctness invariants for both computations with and without interrupts. The concept of correctness without interrupts will only be used in intermediate proof goals, whereas correctness with interrupts actually captures the overall correctness criterion.

Definition 4.3.3 *We call a VAMP implementation configuration* correct without interrupts *in cycle t, i.e., $corr?(t)$, iff the following items hold:*

1. $\tilde{c}_I^t.M = \tilde{c}_S^{sI(M_I,t)}.M$
2. $\tilde{c}_I^t.PC' = \tilde{c}_S^{sI(issue,t)}.PC'$
3. $\tilde{c}_I^t.DPC = \tilde{c}_S^{sI(issue,t)}.DPC$
4. $sI(dec, t) \geq 0 \implies \tilde{c}_I^t.S1.IR = IR(\tilde{c}_S^{sI(dec,t)}) \wedge$
   $\tilde{c}_I^t.S1.imal = imal_S(\tilde{c}_S^{sI(dec,t)}) \wedge$
   $\tilde{c}_I^t.S1.ipf = ipf_S(\tilde{c}_S^{sI(dec,t)})$
5. $\forall x \in \mathbb{Z}_{32} : \tilde{c}_I^t.GPR[x].data = \tilde{c}_S^{sI(wb,t)}.GPR[x]$
6. $\forall x \in \mathbb{Z}_{32} : \tilde{c}_I^t.FPR[x].data = \tilde{c}_S^{sI(wb,t)}.FPR[x]$
7. $\forall x \in \mathbb{Z}_9 : \tilde{c}_I^t.SPR[x].data = \tilde{c}_S^{sI(wb,t)}.SPR[x]$

For the extension to interrupts, we claim less since, e.g., the PC registers may have values never encountered in the specification with interrupts. We can only claim definite values for both PCs and the memory in the cycle immediately after an interrupt. Also note that we only refer to $sI(inst, t)$ since no other scheduling function is defined over interrupt sequences.

Definition 4.3.4 *We call a VAMP implementation configuration* correct with interrupts *in cycle t, i.e., $corr\_i?(t)$, iff the following items hold:*

1. $(t = 0 \vee JISR(c_I^{t-1})) \implies c_I^t.M = c_S^{sI(inst,t)}.M$

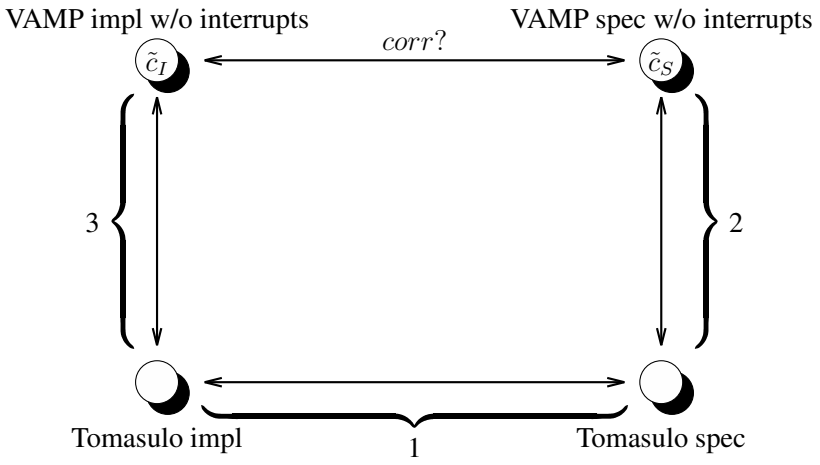

Figure 4.2: Overview of the proof without interrupts

2. $(t = 0 \vee JISR(c_I^{t-1})) \implies c_I^t.PC' = c_S^{sI(inst,t)}.PC'$

3. $(t = 0 \vee JISR(c_I^{t-1})) \implies c_I^t.DPC = c_S^{sI(inst,t)}.DPC$

4. $\forall x \in \mathbb{Z}_{32} : c_I^t.GPR[x].data = c_S^{sI(inst,t)}.GPR[x]$

5. $\forall x \in \mathbb{Z}_{32} : c_I^t.FPR[x].data = c_S^{sI(inst,t)}.FPR[x]$

6. $\forall x \in \mathbb{Z}_9 : c_I^t.SPR[x].data = c_S^{sI(inst,t)}.SPR[x]$

Note that $corr\_i?$ and $spec\_conf$ are by their definition related in the following way:

$$(t = 0 \vee JISR(c_I^{t-1})) \wedge corr\_i?(t) \implies spec\_conf(c_I^t) = c_S^{sI(inst,t)} \quad (4.25)$$

This equation will also become important when we integrate interrupts into the correctness proof.

4.3.3 Proof overview

The overall correctness proof of the VAMP implementation is split into two major parts, correctness without and with interrupts as captured by the two definitions from the previous section. We now want to give an overview over these two steps. We first have a look at correctness without interrupts.

The main idea for showing correctness without interrupts, i.e., predicate $corr?$, is to properly instantiate the Tomasulo correctness proof from [Krö01]. However, this proof is parameterized and generates its own Tomasulo specification and implementation from its parameters. Hence, the proof of $corr?$ is actually split into three parts, the Tomasulo proof and two proofs that the

Tomasulo specification and implementation match their counterparts in the VAMP without interrupts. This is illustrated in figure 4.2. We will now give some more details on the different steps.

1. The key in instantiating the Tomasulo proof lies in the right set of parameters. These parameters can mainly be split into two parts, a specification and an implementation part. The specification part consist of a number of registers, decode functions extracting source and destination register indices from instructions, and a function computing the actual result of an instruction. These parameters are sufficient to generate the Tomasulo specification.

   On the implementation side, we have various functions as parameters that return cycles were, e.g., issue, writeback, and completion occur, or return the results of execution units for any hardware cycle. With the help of all the parameters, the Tomasulo implementation is generated. However, this Tomasulo implementation can obviously only be proved correct with respect to the specification if the above functions fulfill some assumptions, e.g., dispatch can only occur from a full reservation station if all operands are valid and the execution unit does not signal a stall condition. Hence, various rather straightforward assumptions have to be proved in order to instantiate the Tomasulo correctness.

   However, there are also some complex assumptions in the Tomasulo proof like the correctness of execution units and of instruction issuing which actually comprises correctness of instruction fetch that is invisible in the Tomasulo proof. In order to prove these assumptions, we actually need Tomasulo correctness as an assumption, although in a non-circular way. Therefore, we simply instantiate the Tomasulo proof with correct *abstract* execution units and instruction fetch/ issue that fulfill these assumptions by construction, e.g., by just using the corresponding values from the Tomasulo specification as return values of execution units. In the third proof step, when showing that the VAMP without interrupts is actually equivalent to the Tomasulo implementation, these gaps will finally be closed since we can use Tomasulo correctness there. In this thesis, we will not prove any of the straightforward assumptions although they are all in PVS.

2. As a second step, we have to prove equivalence between VAMP specification without interrupts and Tomasulo specifications. We therefore introduce a mapping function between a Tomasulo specification that only contains one large register file and the VAMP specification with three separate register files and show in a straightforward instruction-by-instruction induction proof that this mapping always holds. We will not give any details on this proof part in the rest of the thesis.

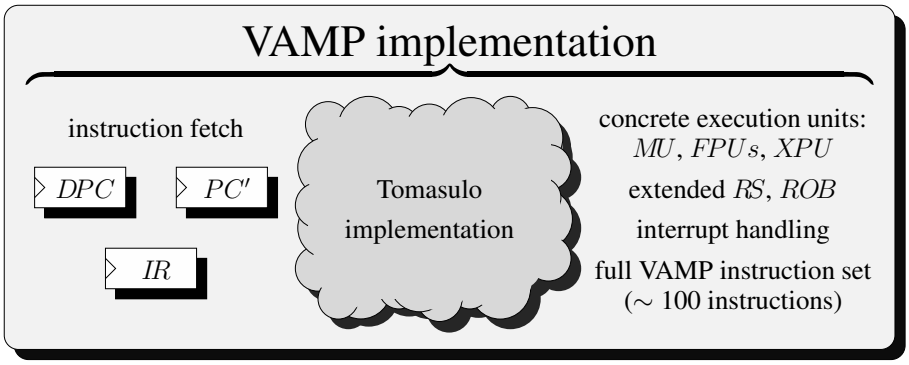

Figure 4.3: VAMP implementation and Tomasulo implementation

3. In the final proof step for correctness without interrupts, we have to match the *Tomasulo implementation* against the VAMP. Note that this Tomasulo implementation as generated by the Tomasulo correctness proof from [Krö01] is just a part of the VAMP as illustrated in figure 4.3 since we have instruction fetch, execution units, and interrupt support in the actual VAMP implementation.

   We introduce a mapping function between the two implementation and show by induction that this mapping always holds. This involves numerous straightforward parts for the induction step, e.g., for the reservation stations, the register file, and the reorder buffer since these registers are simply part of both implementations. We will omit these straightforward parts in this thesis.

   Note that the configuration of the Tomasulo implementation from [Krö01] does not contain some components one would expect, e.g., the destination register indices of instructions in the reorder buffer or the instruction in a reservation station. Instead, these components are 'implemented' with function lookups. Consider, e.g., the destination address for writeback in the reorder buffer. The Tomasulo implementation identifies the instruction in the reorder buffer via its scheduling function and just calls the function returning this instruction's destination indices which is one of the parameters of the proof. Clearly, this procedure cannot by synthesized in hardware. In the VAMP, this function lookup for the destination register is therefore replaced by registers in the *ROB* that are filled during instruction issue and we have to prove that these registers are equivalent to the lookup mechanism of the Tomasulo implementation. Similarly, an instruction register is added to each reservation station such that a function unit actually can decide which operation to perform on dispatch. Both the proof of correctness of the added instruction registers and destination indices is straightforward and therefore omitted in this thesis.

During bus snooping in the reservation stations in the Tomasulo implementation, an embedding function is used which is also a parameter of the proof. This embedding function returns for each register in the register file the index of the result by which it is produced. For multiple results as in the VAMP with double-precision floating point operations, e.g., this embedding function returns 1 for all the registers in the higher half of the floating point register file, and 0 else. Hence, a double precision operand is split among two results, while *both* halves share the same tag since they are produced by the same instruction. During issue and snooping, we therefore need a means to distinguish what part of this result should actually be forwarded. In the Tomasulo implementation, this decision is taken by identifying the instruction via its scheduling function and using the embedding function for this instruction. For the VAMP, we once again need some additional work.

For instruction issue, computing the value of the embedding function is easy since we decode the register index of source operands anyway and can use it to emulate the behaviour of the embedding function. During snooping, we could use the added instruction register in the reservation station to do likewise. However, since we compute the value of the embedding function already during decode, we just extend the reservation station data structure by one additional bit for each source operand and issue the computed value of the embedding function into this additional bit. Basically, this bit extends the tag and tells you which of the results of an instruction is to be forwarded. Once again, after having identified the problem due to the embedding functions, proving that the actual synthesizable VAMP implementation is equivalent to the Tomasulo implementation is straightforward. Note, however, that *any* Tomasulo implementation dealing with mixed single and double precision operations inherently *has to have an extended tag* in any operand of a reservation station for snooping to work correctly. Incidentally, this was one of the bugs we had to fix in the initial VAMP implementation by Kröning [Krö01].

Finally, in some parts of the induction step, we have to fill in the gaps mentioned in the first proof step, i.e., correctness of execution units and instruction fetch. The execution units in the VAMP only work correctly on disjoint subsets of instruction. Therefore, the Tomasulo scheduler has to ensure that, e.g., only memory instruction are issued into the memory reservation station. Since this is also a straightforward proof step, we will not give any details on it in this thesis. Note that the correctness of instruction fetch in particular also entails correctness of the program counters that do not exist in the Tomasulo implementation.

In the following sections, we will therefore focus on these parts of the

induction step. This comprises the result of the memory unit, i.e., the correctness of `load` instructions, and instruction fetch. Note that for the correctness of `load` instructions and instruction fetch, we indirectly also need the correctness of `store` instructions since the result of a `load` depends on memory which is not part of the Tomasulo proof. The correctness of the PCs in the induction step is basically straightforward—the arguments are equal to the ones in the in-order pipeline in [Krö01] since the circuits are just copied and forwarded operands are correct by the Tomasulo correctness invariant we can assume in order to show the induction step. Therefore, we will not give any details on this part of the proof.

Note that for some of the parts of the induction step, we need additional assumptions like the so-called *sync* criterion we will introduce in section 4.4.3 for the correctness of instruction fetch. These assumptions are not needed in the Tomasulo implementation since the corresponding parts are abstracted away and simply replaced by the specification values.

We also integrated the floating point units from [Jac02a] into the VAMP. Unfortunately, the correctness criteria form [Jac02a] do not match the ones employed in the Tomasulo proof and the extension of the Tomasulo proof to the FPU criteria is anything but straightforward. This is due to the fact that the inductive proof of tag uniqueness form [Krö01] has to be completely redone with a modified invariant in order to incorporate the other correctness criteria. Since this involves reproving more than 20 lemmas of the Tomasulo proof from [Krö01], the details are also omitted in this thesis.

Note that the VAMP actually does not implement a strict Tomasulo scheduler with reorder buffer, but a slightly modified version due to its *IEEEf* implementation as discussed in section 4.2.2. Hence, the Tomasulo proof from figure 4.2 is actually an extended proof that also covers this *IEEEf* modification. This necessitates some additional parameters and assumptions for the proof, but the decomposition into proof steps as sketched above is not influenced by this extension. Note also that the additional proof obligations for the new assumptions and the necessary modifications in the induction step are once again straightforward. We actually prove the *IEEEf* extension of the Tomasulo algorithm correct in section 4.4.1.

Now that we have completed the proof overview without interrupts, we will give a short overview over the integration of interrupts that we will discuss in detail in section 4.5. The basic idea is the following: As long as no interrupt occurs, we derive correctness with interrupts, i.e., $corr\_i?$, simply from $corr?$. The interrupt step itself is verified separately. Since the VAMP state after an interrupt is initial, we once again instantiate the correctness proof without interrupt with the actual VAMP after the interrupt

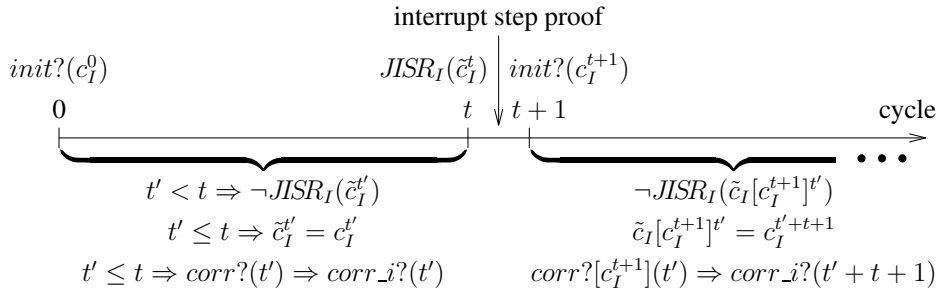

Figure 4.4: Overview of the integration of interrupts into the proof

as initial state. This is illustrated in figure 4.4. Note that the return to an interrupted program is not considered in this part of the proof. This is due to the fact that this return is accomplished by means of a simple `rfe` instructions whose correctness is already established by the correctness proof without interrupts—it is simply a special jump instruction for this proof.

In section 4.5, we will give some details on the interrupt step proof, e.g., that interrupts in the VAMP are really precise and the interrupt implementation matches its specification. Afterwards, we will formalize figure 4.4, i.e., the recursive process of decomposing VAMP computations into interrupt free parts and deriving overall correctness from correctness without interrupts and the interrupt step.

4.4 Correctness without interrupts

In this section, we will focus on the extensions of the Tomasulo correctness proof to the real VAMP architecture without interrupts. This extension consists of the special *IEEEf* implementation, the integration of a memory unit with data access port and instruction fetch from the instruction access port of the memory unit considering self-modification. We basically will show the induction step for the VAMP correctness without interrupts for the above mentioned parts.

We now introduce the main theorem for correctness without interrupt for later use. Since this whole section exclusively deals with the implementation $\tilde{c}_I$ and the specification $\tilde{c}_S$ without interrupts, we omit the explicit $\tilde{c}$ in this section. In the following section about the integration of interrupts, we will again make the distinction between the two different versions explicit.

Lemma 4.4.1 *The VAMP implementation is correct without interrupts. Formally, we have* $\forall t \in \mathbb{N}$*:*

$$synced\_code? \implies corr?(t)$$

Note that we will give a concise definition of the predicate synced_code? in section 4.4.3 when considering instruction fetch.

In the Tomasulo correctness invariant according to [Krö01], a much more complex invariant holds that also claims a special kind of correctness in reservation stations, producers, execution units, and the reorder buffer while not taking memory into account at all. Note that we can use this additional Tomasulo correctness invariant like the induction hypothesis in order to show the induction step.

4.4.1 IEEEf implementation

In [Krö01, p. 304–305], Kröning gives a sketch of the Tomasulo correctness proof with the *IEEEf* implementation we chose in the VAMP. However, this sketch has some huge gaps—it only focuses on writes to *IEEEf* which are more or less straightforward, but ignores the much harder to prove reads from *IEEEf* that require the additional stalling mechanism we had to introduce as a new assumption. Having done the correctness proof with *IEEEf* formally in PVS, we are therefore able to give an accurate overview of the proof. Since about 20 definitions and lemmas are involved in adding this *IEEEf* extension to the proof, we will once again only sketch the highlights.

First of all, we introduce a new specification machine. This machine is equal to the old Tomasulo machine apart from some special register with index idx_{ieeef}. This register is updated in any computation step that does not have idx_{ieeef} as destination register by the result of some function f_{ieeef} on the old value of register idx_{ieeef} and the regular results of the instruction. Note that for the VAMP case, this function f_{ieeef} is given by the second part of equation (4.8) from the definition of the programmer's model.

The new implementation only differs from the standard Tomasulo implementation in one little detail: If an instruction does not write register idx_{ieeef} during writeback, the new value of register idx_{ieeef} is computed by applying f_{ieeef} to the current value of register idx_{ieeef} in the register file and the results of the instruction in the reorder buffer. As we mentioned before, we also have the additional assumptions that instructions with source operand idx_{ieeef} are only issued if the reorder buffer is empty.

Note that the data consistency invariant 6.1 from [Krö01, p. 287] for the register file in the standard Tomasulo algorithm has to be changed. Kröning claimed that if a register in the register file is valid, it contains the data the next instruction to be issued would read as a source operand. However, this invariant no longer holds for register idx_{ieeef}. Therefore, we split the invariant into the old part for any register but idx_{ieeef} and add a new part by claiming that register idx_{ieeef} *always* contains the data the instruction currently in the *writeback* stage is supposed to read as source operand—independent of the valid bit of idx_{ieeef}.

Because of the additional assumption that issue of an instruction reading idx_{ieeef} can only occur if the reorder buffer is empty, i.e., if the scheduling functions for issue and writeback are equal, the new invariant guarantees the correctness of reads from idx_{ieeef} during issue. Note that because of the additional assumption for idx_{ieeef}, no reservation station has to snoop for idx_{ieeef} since the source operand idx_{ieeef} in a reservation station is always valid. Recall that for the VAMP, register idx_{ieeef} is not the Tomasulo source operand of any floating point instruction, but only for special moves `movs2i` with source *IEEEf*. This is also crucial since correctness of the standard Tomasulo forwarding mechanism employed during snooping for the reservation stations cannot be applied to register idx_{ieeef}.

During instruction writeback, we have to take special care of register idx_{ieeef}. However, these arguments are basically straightforward: If an explicit write to idx_{ieeef} occurs, the correctness follows from the old proof. On an implicit write by means of a floating point instruction, we have that the new value of idx_{ieeef} is computed by applying f_{ieeef} to the current value and the result in the *ROB*. The result of the instruction in the reorder buffer is already correct by invariant. The new version of invariant 6.1 additionally guarantees the correctness of the current value of idx_{ieeef} since we are considering the instruction in stage writeback. Hence, implicit updates by floating point instructions also compute the correct value and we have the correctness of an extended Tomasulo algorithm.

Applying this proof to the actual VAMP requires only two things, namely showing that a `movs2i` with source register *IEEEf* is stalled in decode until the *ROB* is empty and defining the function f_{ieeef} needed in instantiating the proof. Note that we already gave this definition as part of the definition of the next value of Register *IEEEf* in a computation without interrupts in equation (4.8).

An interesting side effect of the *IEEEf* implementation is the following: Consider a `movs2i` instruction with source *IEEEf* and destination register 0 in the *GPR*. This instruction is just a `nop` in the view of the programmer's model since register 0 always returns 0. However, because of the *IEEEf* implementation, the VAMP is stalled until it has run empty before the instruction leaves stage decode and a new instruction is fetched. Hence, this instruction has some synchronizing effect which we will exploit when dealing with self-modifying code in section 4.4.3. We therefore call a `movs2i` with source *IEEEf* and destination 0 `sync` instruction.

$$\texttt{sync}?(c_S) := \texttt{movs2i}?(c_S) \wedge SA(c_S) = 7 \wedge RD(c_S) = 0 \tag{4.26}$$

4.4.2 VAMP memory unit

In this section, we describe the implementation and verification of the VAMP memory unit. As the main component of the memory unit, we have a memory interface as defined in section 1.5.1. The VAMP offers byte, half word,

word, and double word accesses to the data memory, i.e., we support a width d of the memory operand in bytes with $d \in \{1, 2, 4, 8\}$. For memory operations and instruction fetch, the VAMP supports 32-bit byte addresses. Hence, we have a 4 GB byte-addressed memory in the specification. In the implementation, however, we want to fit any memory operation of the VAMP into a single access to the memory interface. We therefore use a memory interface with a data word width of the maximum access width of 8 bytes, i.e., we instantiate a correct memory interface according to definition 1.5.2 with $a = 29$ and $B = 8$ and assume little-endian memory organization. This means that the lowest byte of a data word is stored at the lowest byte address, e.g., when storing the word $0{\times}$a1b2c3d4 at address ad, byte $d4$ is stored at address ad, $c3$ at address $ad + 1$, and so on.

Note also that the VAMP supports both signed and unsigned operations for read accesses, e.g., in case of a signed byte load, sign extension of the read byte data is performed in the memory unit. In the following, we first show how any VAMP memory operation is mapped to *one* double word access to memory interface. We then describe the actual hardware in the memory unit and finish with the correctness proof for load and store instructions.

Mapping VAMP memory accesses to a memory interface

Let ea be the effective byte address of a memory access as introduced in equation (4.2). Memory accesses are required to be aligned in the VAMP, i.e., $ea \bmod d = 0$ must hold. Otherwise, an exception is raised without accessing the memory at all, i.e., we have $dmal = (ea \bmod d \neq 0)$ and $EData = ea$ as introduced in section 4.1. Because of the alignment restriction, one can easily prove that any data memory access of the VAMP can be realized by a single access to a double-word organized 4 GB memory, i.e., a memory interface with $B = 8$ and $a = 29$.

Lemma 4.4.2 *An aligned memory access to an address $ea \in \mathbb{Z}_{2^{32}}$ is simply an access to bytes $ea \bmod 8$ to $(ea \bmod 8) + d - 1$ of the double word address $ea \operatorname{div} 8$.*

Proof: We trivially know $8 \cdot (ea \operatorname{div} 8) + (ea \bmod 8) = ea$. Because of the little-endian memory organization, the byte at address ea can be found at byte $ea \bmod 8$ in the double word at address $ea \operatorname{div} 8$. We thus only have to show that all d bytes have the double word address $ea \operatorname{div} 8$, i.e., that the access does not span two double words. Because of $ea \bmod d = 0$, we conclude $d|ea$ and thus, for any $j \in \mathbb{Z}_d$, we have

$$(ea + j) \operatorname{div} d = ea \operatorname{div} d$$

and since $d|8$ holds, this leads to

$$(ea + j) \operatorname{div} 8 = ea \operatorname{div} 8$$

and the proof is finished since all d bytes have the double word address ea div 8. □

The data input and output of a double-word organized memory is obviously 64 bits wide. In order to embed memory accesses of width $d < 8$, the data to be written has to be shifted to the correct byte position in a double word of the memory interface. According to the above lemma, the corresponding shift distance equals ea mod 8 bytes, i.e. $8 \cdot (ea \bmod 8)$ bits. We call a circuit shifting data ea mod 8 bytes to the left *shift4store*. A similar shifting has to be done for read accesses, i.e., the correct bytes have to be extracted from the 64 bit data word of the memory interface. Thus, a circuit shifting a double word ea mod 8 bytes to the right and only returning the d rightmost bytes of the result is called *shift4load*. Since both signed and unsigned loads are supported, we have to perform sign-extension or zero-extension of loaded data, respectively, which is achieved by circuits *sext* or *zext*. These circuits can be integrated into *shift4load*, i.e., depending on an additional input, *shift4load* returns either the sign- or zero-extended shifted data.

For store operations, we additionally have to generate the correct byte write signals *mwb* for the memory interface access. We call a circuit generating the byte write signals for byes ea mod 8 to $(ea \bmod 8) + d - 1$ in the memory interface *genbw*.

Definition 4.4.3 *We first specify* shift4load *and* shift4store *for* $d \leq 4$.

$$\begin{aligned} sh4s'(d, off, din) &:= din[8 \cdot (4 - off) - 1 : 0]din[31 : 8 \cdot (4 - off)] \\ sh4l'(s, d, off, din) &:= \begin{cases} sext(din[8 \cdot (off + d) - 1 : 8 \cdot off]) & \text{if } s = 1 \\ zext(din[8 \cdot (off + d) - 1 : 8 \cdot off]) & \text{otherwise} \end{cases} \end{aligned}$$

For and arbitrary $d \leq 8$, *we set* $off' := off \bmod 4$ *and*

$$din' := off \text{ div } 4 = 0 ? din[31 : 0] : din[63 : 32]$$

and have the general specifications for the circuits shift4load, shift4store, *and* genbw.

$$\begin{aligned} sh4s(d, off, din) &:= \begin{cases} din & \text{if } d = 8 \\ sh4s'(d, off', din')sh4s(d, off', din') & \text{otherwise} \end{cases} \\ sh4l(s, d, off, din) &:= \begin{cases} din & \text{if } d = 8 \\ sh4l(s, d, off', din')sh4l(s, d, off', din') & \text{otherwise} \end{cases} \\ genbw(d, off) &:= \lambda_{i \in \mathbb{Z}_8} i \geq off \wedge i < off + d \end{aligned}$$

Lemma 4.4.4 *With the help of the above definitions, we can replace a byte oriented memory with a double word memory. Formally, let us consider*

an aligned memory instruction with effective memory address $ea \in \mathbb{Z}_{32}$*, a byte width of* $d \in \{1, 2, 4, 8\}$*,* $s := \texttt{lb?} \vee \texttt{lh?}$ *as flag for a signed access, and* $m := M[8 \cdot (ea \text{ div } 8) + 7 : 8 \cdot (ea \text{ div } 8)]$ *the double word addressed by* $ea \text{ div } 8$*. The result of any load operation according to tables A.1 and A.4 is then given by* $sh4l(s, d, ea \bmod 8, m)$*. Similarly, writing bytes* $genbw(d, ea \bmod 8)$ *of* $sh4s(d, ea \bmod 8, data)$ *in the double word address* $ea \text{ div } 8$ *is the specified effect of an aligned store operation.*

Proof: We split cases on the type of the memory operation.

1. Let us consider a load instruction. For $d = 8$, we have $ea \bmod 8 = 0$ because of the alignment and hence, $8 \cdot (ea \text{ div } 8) = ea$. This leads to

$$\begin{aligned} sh4l(s, d, ea \bmod 8, m) &= m \\ &= M[8 \cdot (ea \text{ div } 8) + 7 : 8 \cdot (ea \text{ div } 8)] \\ &= M[ea + 7 : ea] \end{aligned}$$

   and concludes the case $d = 8$. For $d \leq 4$, we are only interested in the 4 lowest bytes of $sh4l$. Note that $(ea \bmod 8) \text{ div } 4 = 0$ holds iff $ea \text{ div } 4 = 0$. In addition, we know

$$4 \cdot (ea \text{ div } 4) = \begin{cases} 8 \cdot (ea \text{ div } 8) & \text{if } ea \text{ div } 4 = 0 \\ 8 \cdot (ea \text{ div } 8) + 4 & \text{otherwise} \end{cases}$$

   Hence, the din' input of $sh4l'$, i.e., $ea \text{ div } 4 = 0?\, m[31:0] : m[63:32]$ equals $m' := M[4 \cdot (ea \text{ div } 4) + 3 : 4 \cdot (ea \text{ div } 4)]$ by the definition of m.

$$sh4l(s, d, ea \bmod 8, m)[31:0] = sh4l'(s, d, ea \bmod 4, m')$$

   By expanding the definition of $sh4l'$, we get

$$= \begin{cases} sext(m'[8 \cdot ((ea \bmod 4) + d) - 1 : 8 \cdot (ea \bmod 4)]) & \text{if } s = 1 \\ zext(m'[8 \cdot ((ea \bmod 4) + d) - 1 : 8 \cdot (ea \bmod 4)]) & \text{otherwise} \end{cases}$$

   Since $ea = 4 \cdot (ea \text{ div } 4) + (ea \bmod 4)$ trivially holds, we also have $m'[8 \cdot ((ea \bmod 4) + d) - 1 : 8 \cdot (ea \bmod 4)] = M[ea + d - 1 : ea]$ which equals the definition of the effect of a load instruction in appendix A and this case is finished.

2. For store operations, we omit the proof due to its similarity to the above case of load operations. □

The design of circuits implementing *s4l* with integrated *sext* and *zext*, *s4s*, and *genbw* according to the above specifications is straightforward and the corresponding correctness proofs are also easy. A sample implementation of these circuits can be found in [MP00, p. 78–88].

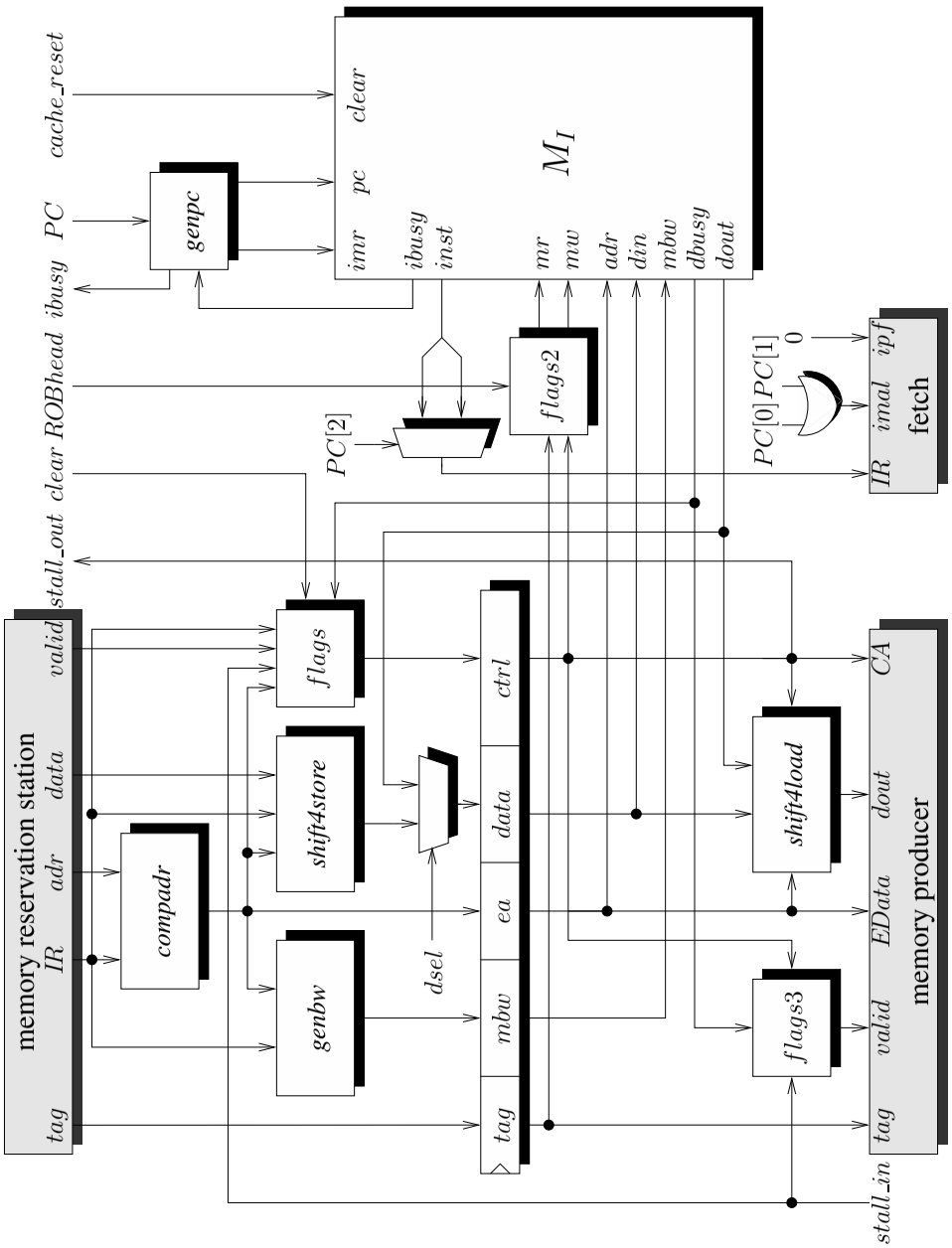

Figure 4.5: The VAMP memory unit

Hardware of the memory unit

The memory unit of the VAMP basically consists of the circuits specified in the last paragraph, one pipeline stage, and a memory interface as depicted in figure 4.5. On dispatch, the effective address ea is computed from the instruction register and the address in the reservation station and written into the memory stage together with the data shifted for stores, the generated byte write signals, the tag from the reservation station, and several flags encoding the type of memory access, and the access width d. The instruction in the memory stage can then access the memory interface—unless there was a misalignment and no memory access is needed at all. Note that stores are required to wait with their memory access until their tag equals $ROBhead$. The access to the memory interface may take several cycles. During this time, the memory stage is stalled and $stall\_out$ is signalled to the Tomasulo scheduler in order to signal that the unit can currently accept no further instructions by dispatch.

When the memory interface finally signals the end of the access with $\neg c_I.M.dbusy$, there are two possibilities. Either the Tomasulo scheduler signals by $stall\_in$ that it can currently accept no outputs from the memory unit or it does not. In case of $\neg stall\_in$, the instruction leaves the memory unit and enters the producer register. In case of a load instruction, this additionally means that the data output $dout$ of the memory interface is passed to circuit *shift4load* and the output, the actual result of the instruction, is written to the producer register. If, on the other hand, $stall\_in$ holds at the end of the memory access, a flag in the memory stage is asserted signalling that the instruction in the memory unit already accessed the memory unit, but is only waiting for $stall\_in$ to be deactivated. In case of a read access to the memory interface, the data returned by the memory interface is stored in the $data$ register of the memory stage. As soon as $stall\_in$ is deactivated, the instruction leaves the memory unit; for load instructions, $data$ is passed to *shift4load* in order to compute the result of the instruction.

Additionally, instruction fetch is performed in the memory unit. Therefore, there is an additional input PC which is actually used for instruction fetch. A definition of this PC will be given in the following section 4.4.3 about instruction fetch. Since we have a double-word wide memory interface, we use $PC[2]$ in order to select which half of the 64 bit data word is returned in order to be saved to the instruction register.

Note that we have to guarantee valid input to the memory according to definition 1.5.1. It is easy to guarantee that on the data port, mr and mw are never raised simultaneously. Input stability for the effective address in the memory stage is also easy to guarantee in the absence of interrupts. On an interrupt, however, the PC may be forced to a new value in the middle of an access and the memory stage is flushed.

For the memory stage, this problem is easy to solve. We add an addi-

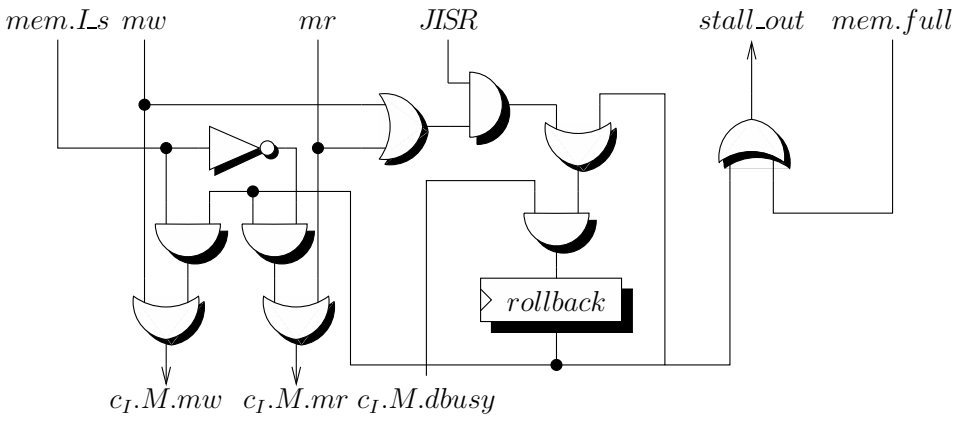

Figure 4.6: Stabilizing circuit for a data access in the *MU*

tional control bit *rollback* to the memory stage that is only asserted when a $JISR(c_I)$ occurs on a pending memory access according to figure 4.6.[1] The memory read- and write inputs of the memory interface are then also activated on an active *rollback* signal; the type of the access is encoded in a I_s bit which is active for store operations. The *rollback* control bit stays active until the memory interface signals $\neg c_I.M.dbusy$. In this case, the memory stage is just flushed and the instruction in it *does not leave the memory unit.* In addition, the memory unit has to signal $stall\_out$ as long as this control bit is asserted. The following equations summarize these computations as depicted in figure 4.6.

$$\begin{array}{rcl} rollback' & = & ((mr_I \vee mw_I) \wedge JISR(c_I) \vee rollback) \wedge c_I.M.dbusy \\ stall\_out & = & mem.full_I \vee rollback \\ c_I.M.mr & = & mr_I \vee rollback \wedge \neg mem.I\_s \\ c_I.M.mw & = & mw_I \vee rollback \wedge mem.I\_s \end{array} \tag{4.27}$$

For the PC, we have to add an additional register mPC that stores the PC at the first cycle of an access according to figure 4.7; the memory interface is accessed with mPC from the second cycle on. If the memory interface signals $\neg c_I.M.ibusy$, we compare the current PC with mPC. In case of equality, we pass $\neg c_I.M.ibusy$ to the instruction fetch; otherwise, we signal *ibusy* to the instruction fetch and start a new fetch with the current PC in the next cycle. This additional register in order to keep the PC input to the instruction memory stable is not realized in [MP00] which once again shows the necessity of formally putting all the proofs together in order to really obtain overall correctness. Note that this does not impact on VAMP liveness since this address change can only occur on a $JISR(c_I)$ and there

[1] Note that we will verify later on that store accesses cannot be interrupted by $JISR(c_I)$.

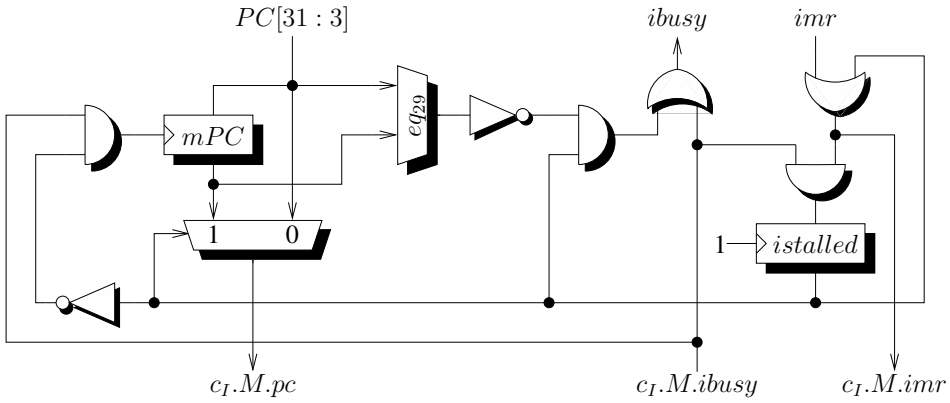

Figure 4.7: PC stabilizing circuit *genPC*

can be no two cycle where $JISR(c_I)$ is active without at least one instruction fetch happening in between. The following equations just summarize the schematics from figure 4.7.

$$\begin{aligned}
installed' &= (imr \vee installed) \wedge c_I.M.ibusy \\
mPC' &= \begin{cases} PC[31:3] & \text{if } c_I.M.ibusy \wedge \neg installed \\ mPC & \text{otherwise} \end{cases} \\
c_I.M.imr &= imr \vee installed \\
c_I.M.pc &= \begin{cases} mPC & \text{if } installed \\ PC[31:3] & \text{otherwise} \end{cases} \\
ibusy &= c_I.M.ibusy \vee installed \wedge \neg eq_{29}(PC[31:3], mPC)
\end{aligned} \tag{4.28}$$

Correctness of the memory unit

We have to show that the memory stage in cycle t produces as output the corresponding result from the programmer's model, i.e., the correct data is read in case of a load operation. Additionally, we have to show that the memory in the following cycle $t+1$ still fulfills the correctness invariant, i.e., the correctness of store. In both these proofs, we can assume $corr?(t)$ and Tomasulo correctness up to cycle t. As part of the Tomasulo correctness, we have correctness of the operands and the instruction registers in the memory reservation station. It is therefore easy to derive correctness for the registers in the memory stage, i.e., the effective address, several flags, and the data to be stored.

For the correctness of loads, we only focus on the cycle a load actually completes. Proving that this data is stored in the memory stage and passed to the producer later on an active *stall_out* of the Tomasulo scheduler is straightforward. In addition, we only focus on aligned accesses since

the correctness of a misaligned access is easy to show. We start with a helper lemma stating that the specification memory that the instruction in the memory stage sees at the end of an access is actually the specification memory of instruction $sI(M_I, t)$.

Lemma 4.4.5 *If an instruction in the memory unit terminates its access in cycle t, the specification memory for the instruction in the memory unit and the one of the instruction identified by the visible memory register scheduling function are identical. Formally, we have*

$$(mw_I^t \vee mr_I^t) \wedge \neg\tilde{c}_I^t.M.dbusy \implies \tilde{c}_S^{sI(M_I,t)}.M = \tilde{c}_S^{sI(mem,t)}.M$$

Proof: Let $(mw_I^t \vee mr_I^t) \wedge \neg\tilde{c}_I^t.M.dbusy$ hold. We set $i := sI(mem, t)$ and have to show $\tilde{c}_S^{sI(M_I,t)}.M = \tilde{c}_S^i.M$. We can prove $sI(M_I, t) \leq sI(mem, t) + 1$ by trivial induction on t with the definition of $sI(M_I, t)$ and the fact that $sI(mem, t)$ is increasing on t. In addition, we can conclude that $sI(M_I, t) = i + 1$ can only hold if there is some cycle $t' < t$ with $(mw_I^{t'} \vee mr_I^{t'}) \wedge \neg\tilde{c}_I^{t'}.M.dbusy$ and $sI(mem, t') = i$. However, since memory instructions access the data memory only once and instruction i accesses the memory in cycle t, this cannot be the case and we therefore know $sI(M_I, t) \leq i$. It is therefore sufficient to show that no instruction j, $sI(M_I, t) \leq j < i$ writes the data memory. We show the claim by contradiction, i.e., we assume that we have and instruction j with

$$\neg dmal(\tilde{c}_S^j) \wedge (\texttt{sw}? \vee \texttt{sh}? \vee \texttt{sb}? \vee \texttt{fstore}?)(\tilde{c}_S^j)$$

Since instruction i accesses the memory in cycle t and the memory is accessed in order, we find a cycle $t' < t$ where instruction j was in the memory unit, i.e., $sI(mem, t') = j$, and since $\neg dmal(\tilde{c}_S^j)$ additionally holds, we can fix this t' such that the instruction actually access the data memory, i.e., $mw_I^{t'} \wedge \neg\tilde{c}_I^{t'}.M.dbusy$ holds. In this case, we have

$$sI(M_I, t' + 1) = sI(mem, t') + 1 = j + 1 > sI(M_I, t)$$

which is a contradiction since $t' + 1 \leq t$ holds and the scheduling function $sI(M_I, t)$ is increasing on t. □

Lemma 4.4.6 *Let the CPU correctness invariant from definition 4.3.3 hold in cycle t and let a load instruction in the memory unit complete its aligned access in cycle t. This load instruction then delivers the correct result, i.e., we set $i := sI(mem, t)$ and have $corr?(t) \wedge mr^t \wedge \neg\tilde{c}_I^t.M.dbusy \implies$*

$$MU.dout^t = \begin{cases} sext(\tilde{c}_S^i.M[ea(\tilde{c}_s^i) + d - 1 : ea(\tilde{c}_s^i)]) & \text{if } (\texttt{lb}? \vee \texttt{lh}?)(\tilde{c}_S^i) \\ zext(\tilde{c}_S^i.M[ea(\tilde{c}_s^i) + d - 1 : ea(\tilde{c}_s^i)]) & \text{otherwise} \end{cases}$$

Proof: Let a load instruction in the memory unit finish its memory access in cycle t, i.e., $mr^t \wedge \neg \tilde{c}_I^t.M.dbusy$ holds. By lemma 4.4.5, $\tilde{c}_S^i.M = \tilde{c}_S^{sI(M_I,t)}.M$ holds in cycle t. Thus, $\tilde{c}_I^t.M = \tilde{c}_S^i.M$ holds because of $corr?(t)$. It is easy to conclude correct data in the memory stage, in particular $ea(\tilde{c}_I^t) = ea(\tilde{c}_S^i)$. The correctness of the memory interface additionally guarantees $mem.dout^t = \tilde{c}_I^t.M[\langle ea(c_S^i)\rangle + 7 : \langle ea(c_S^i)\rangle]$. Since *MU.dout* is given by the output of circuit *shift4load*, the access is aligned, and the flags in the memory stage are correct, lemma 4.4.4 concludes the proof. □

Lemma 4.4.7 *Let the CPU correctness invariant hold in cycle t. The memory part of the invariant still holds in cycle $t+1$, i.e., $\forall t \in \mathbb{N}$:*

$$corr?(t) \implies \tilde{c}_I^{t+1}.M = \tilde{c}_S^{sI(M_I,t+1)}.M.$$

Proof: Because of $corr?(t)$, we conclude $\tilde{c}_I^t.M = \tilde{c}_S^{sI(M_I,t)}.M$. If no instruction finishes its memory access in cycle t, $sI(M_I, t+1) = sI(M_I, t)$ and $\tilde{c}_I^{t+1}.M = \tilde{c}_I^t.M$ both hold and the claim is finished because of $corr?(t)$. Let therefore an instruction finish its memory access in cycle t, i.e., we have $(mw_I^t \vee mr_I^t) \wedge \neg \tilde{c}_I^t.M.dbusy$. In this case, we conclude $sI(M_I, t+1) = sI(mem, t) + 1$ and lemma 4.4.5 together with $corr?(t)$ guarantees $\tilde{c}_I^t.M = \tilde{c}_S^{sI(mem,t)}.M$. If mw_I^t does not hold, we have both $\tilde{c}_S^{sI(mem,t)+1}.M = \tilde{c}_S^{sI(mem,t)}.M$ and $\tilde{c}_I^{t+1}.M = \tilde{c}_I^t.M$ and are also finished. Let therefore mw_I^t hold. Since the memory stage contains correct data for instruction $sI(mem, t)$, the byte write signals and the data shifted for store are correct in particular. Lemma 4.4.4 then concludes the proof. □

Note that in both above lemmas, we omitted the case of misaligned accesses; this is due to the fact that detecting $ea \bmod d \neq 0$ in the implementation and reacting by simply not accessing the memory, but signalling misalignment is easy.

4.4.3 Instruction fetch

For instruction fetch, we have to deal with self-modifying code since an instruction that has already been fetched could be overwritten by a store instruction still in the pipeline. There are basically four solutions to the self-modification problem:

1. Completely forbid writes to the instruction stream. This solution is not realistic since, e.g., an operating system usually wants to start an application it has loaded previously into memory, i.e., instructions have to be fetched from memory locations that have been written previously.

2. Compute an architecture-dependent constant min_i such that instruction fetch always works correct if there are at least min_i instructions between a write into the instruction stream and the first fetch of a

modified address. This constant min_i is somehow derived from the maximum number of instructions that can be simultaneously in the pipeline. In other words, a future generation of a CPU that basically doubles the number of pipeline stages is no longer fully compatible to the previous version because min_i increases due to the higher number of instructions simultaneously in the pipeline. We therefore discard this solution.

3. Handle self-modification as a form of speculation. If a write to an instruction currently being fetched or already in the pipeline completes, we trigger a misspeculation which causes a rollback starting at the instruction that was overwritten. This overwritten instruction is then fetched again and since the write that caused the speculation rollback is now completed, this new fetch already reads the freshly written data. This solution requires a considerable additional effort regarding verification; for the implementation, we basically have to compare the memory address of a store instruction with all PCs in the pipeline. On the other hand, this solution delivers the cleanest model: The implementation simulates the specification without any code restrictions. However, since self-modification is basically only used when switching back from operating system to a loaded user application, this solution seems somewhat excessive and we do not investigate it further.

4. Require the usage of a `sync` instruction before a modified fetch. No matter how far apart a write to the instruction stream and the modified fetch, we require that there is a `sync` instruction in between. This solution is architecture-independent and requires very little hardware overhead. Since typical compilers do not produce self-modifying code, compiled programs usually fulfill the *sync* criterion by default. Basically, the only thing an operating system has to guarantee then is that a switch back to a user thread only occurs after a `sync` instruction. We therefore focus on this solution for self-modifying code.

Let us assume that a `sync` instruction in the CPU has the following effects:

1. From the programmer's view, a `sync` instruction behaves like a `nop`.

2. On the implementation side, a `sync` instruction prohibits any further instruction fetch until all instructions sequentially before the `sync` instruction have left the pipeline.

As we already sketched in section 4.4.1, there is already a special version of a `movs2i` instruction in the VAMP fulfilling these criteria; we called it simply `sync`. First of all, we formalize the above idea of requiring `sync` instruction before modified fetches.

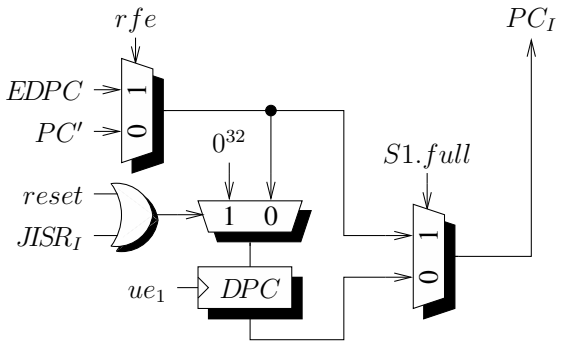

Figure 4.8: Fetch PC implementation in the VAMP

Definition 4.4.8 *We call assembly code synced_code? iff for any address ad there is a* `sync` *instruction between the fetch of ad and the last modification of address ad. We introduce a parameterized predicate* $write_{ad}$ *on a specification configuration* c_S *that holds iff a write to address ad occurs in* c_S*, i.e., the effective address matched ad and either a store or a floating point store operation occur. Formally, we thus have*

$$write_{ad}(c_S) :\Longleftrightarrow (ea(c_S) = ad \bmod 8) \wedge (\texttt{store}?(c_S) \vee \texttt{fstore}?(c_S)).$$

Note that since $write_{ad}$ *is a predicate on a configuration, we can apply definition 1.2.8 in order to argue about the last write of a computation. Since we distinguish two different computations depending on whether we react to interrupts, we have to use the prefix* c_S *or* $\tilde{c}_S$ *in order to distinguish between the two possible instantiations.*

A computation without interrupts fulfills the sync condition if the following condition holds:

$$\begin{aligned} synced\_code? \;:\Longleftrightarrow\; & \forall n \in \mathbb{N} : \tilde{c}_S.\exists^{last}_{write_{\tilde{c}^n_S.DPC}} \Longrightarrow \\ & \exists m \in]\tilde{c}_S.last_{write_{\tilde{c}^n_S.DPC}} : n[: \\ & \quad \texttt{sync}?(\tilde{c}^m_S) \end{aligned}$$

From the view of an assembly program, interrupts are in general nondeterministic, e.g., timer-interrupts. Therefore, we do not demand that assembly programs fulfill the sync condition across interrupt calls, but only that the sync criterion without interrupts be fulfilled for any starting state that can be reached in the specification with interrupts.

$$synced\_code\_i? :\Longleftrightarrow \forall n \in \mathbb{N} : synced\_code?[c^n_S]$$

Before we start with the actual correctness lemma for instruction fetch, we introduce a simple lemma stating the correctness of the PC used in the VAMP for fetch operations. This will abstract the main proof from the

delayed PC architecture. We first define a function that actually returns the PC used in the implementation for instruction fetch which is based on [Krö01]. Figure 4.8 shows the corresponding schematics.

$$PC(c_I) := \begin{cases} c_I.SPR[EDPC] & \text{if } c_I.S1.full \wedge \texttt{rfe}?(c_I) \\ c_I.PC' & \text{if } c_I.S1.full \wedge \neg\texttt{rfe}?(c_I) \\ c_I.DPC & \text{otherwise} \end{cases} \tag{4.29}$$

Lemma 4.4.9 *Let the correctness invariant hold in cycle t and a fetch occur. The fetch PC is then correct, i.e., we have $\forall t \in \mathbb{N}$:*

$$corr?(t) \wedge \neg stall.1^t \implies PC(\tilde{c}_I^t) = \tilde{c}_S^{sI(fetch,t)}.DPC.$$

Proof: We set $i := sI(fetch, t)$ and split cases on $S1.full^t$.

1. Let $S1.full^t$ hold. This leads to $sI(dec, t) = i - 1$ and $sI(issue, t) = i-1$. By expanding the definition of $PC(\tilde{c}_I^t)$, we therefore only have to show $\tilde{c}_I^{t\prime}.DPC = \tilde{c}_S^{i-1\prime}.DPC$. Since $S1.full^t$ and $\neg stall.1^t$ both hold, we know that all forwarded source operands of the instruction in the decode stage are correct by the Tomasulo correctness in cycle t, in particular $\tilde{c}_I^t.EDPC = \tilde{c}_S^{i-1}.EDPC$, and $corr?(t)$ additionally guarantees $\tilde{c}_I^t.IR = \tilde{c}_S^{i-1}.IR$ and $\tilde{c}_I^t.PC' = \tilde{c}_S^{i-1}.PC'$. We conclude the proof by expanding the definition of the primed versions.

$$\tilde{c}_S^{i-1\prime}.DPC = \begin{cases} c_S^{i-1}.SPR[EDPC] & \text{if } \texttt{rfe}?(c_S^{i-1}) \\ c_S^{i-1}.PC' & \text{otherwise} \end{cases}$$

$$\tilde{c}_I^{t\prime}.DPC = \begin{cases} c_I^t.SPR[EDPC] & \text{if } \texttt{rfe}?(c_I^t) \\ c_I^t.PC' & \text{otherwise} \end{cases}$$

2. Let $\neg S1.full^t$ hold. This leads to $sI(issue, t) = i$ and because of $corr?(t)$, we can conclude

$$PC(\tilde{c}_I)^t = \tilde{c}_I^t.DPC = \tilde{c}_S^i.DPC$$

and finish the lemma. □

With the help of this lemma, we are independent of the delayed PC architecture for the following proof since we have the correctness of the PC used for instruction fetch.

Lemma 4.4.10 *Let the VAMP fulfill the correctness invariant in cycle t and let the sync condition on the assembler code hold. The instruction register part of the correctness invariant then holds in cycle $t + 1$, i.e., we set $i := sI(dec, t + 1)$ and have $\forall t \in \mathbb{N}$:*

$$\begin{aligned} &synced\_code? \wedge corr?(t) \wedge i \geq 0 \implies \\ &\tilde{c}_I^{t+1}.S1.IR = IR(\tilde{c}_S^i) \wedge \\ &\tilde{c}_I^{t+1}.S1.imal = imal(\tilde{c}_S^i) \wedge \tilde{c}_I^{t+1}.S1.ipf^{t+1} = CA(\tilde{c}_S^i)[3] \end{aligned}$$

Proof: Let *synced_code*?, *corr*?(t), and $i \geq 0$ hold. If no fetch occurs in cycle t, i.e., $stall.1^t \vee ibusy^t$, we have $i = sI(dec, t)$, $\tilde{c}_I^{t+1}.S1.IR = \tilde{c}_I^t.S1.IR$, $\tilde{c}_I^{t+1}.S1.imal = \tilde{c}_I^t.S1.imal$, $\tilde{c}_I^{t+1}.S1.ipf = \tilde{c}_I^t.S1.ipf$, and the proof is finished because of the instruction register part of *corr*?(t) holds. Hence, we assume that a fetch occurs, i.e., $\neg stall.1^t \wedge \neg ibusy^t$. This leads to $i = sI(dec, t) + 1$ and $i = sI(fetch, t)$. With lemma 4.4.9, we additionally ensure that $PC(\tilde{c}_I^t) = \tilde{c}_S^i.DPC$ holds. If $\tilde{c}_S^i.DPC$ is misaligned, the proof is trivially finished. Since there is no address translation in the VAMP, page faults are tied to 0. Hence, we only have to show the above claim for the instruction register itself.

Let us therefore assume an aligned PC, i.e., we have $\tilde{c}_S^i.DPC \bmod 4 = 0$. Note that the instruction port of the memory interface is accessed on address $pc := PC(c_I^t) \operatorname{div} 8$ and depending on $PC(c_I^t) \bmod 8$, either the upper or lower word is selected as input to the instruction register. Because of $\neg ibusy^t$, definition 1.5.2 of a correct memory interface, and *corr*?(t), we have

$$\begin{aligned}
\tilde{c}_I^{t+1}.S1.IR &= \begin{cases} \tilde{c}_I^t.M[8 \cdot pc + 7 : 8 \cdot pc + 4] & \text{if } \tilde{c}_S^i.DPC \bmod 8 = 4 \\ \tilde{c}_I^t.M[8 \cdot pc + 3 : 8 \cdot pc + 0] & \text{otherwise} \end{cases} \\
&= \tilde{c}_I^t.M[8 \cdot pc + (\tilde{c}_S^i.DPC \bmod 8) + 3 : \\
&\qquad 8 \cdot pc + (\tilde{c}_S^i.DPC \bmod 8)] \\
&= \tilde{c}_I^t.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] \\
&= \tilde{c}_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC]
\end{aligned}$$

Since $IR(\tilde{c}_S^i) = \tilde{c}_S^i.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC]$ holds, the claim is finished if

$$\tilde{c}_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] = \tilde{c}_S^i.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC]$$

holds, and we can assume inequality in order to construct a contradiction. Note that since instruction i is fetched into the pipeline in cycle t, $i \geq sI(M_I, t)$ holds which actually requires a separate induction proof which we omit due to its simplicity. According to definition 4.4.8, we therefore find an instruction j with $sI(M_I, t) \leq j < i$ and $write_{\tilde{c}_S^i.DPC}(\tilde{c}_S^j)$, i.e., $\tilde{c}_S.\exists^{last}_{write_{\tilde{c}_S^i.DPC}}$ holds. We set $l := \tilde{c}_S.last_{write_{\tilde{c}_S^n.DPC}}$ and we additionally have $l \geq sI(M_I, t)$. Thus, *synced_code*? guarantees `sync`?$(\tilde{c}_S^k)$ for some $k \in]l : i[$.

Because of $k < i$, instruction k has already been fetched. Since a new instruction is fetched in cycle t, i.e., $\neg stall.1^t \wedge \neg ibusy^t$ holds, this `sync` instruction k has either already left stage decode or is about to do so. However, the implementation of the `sync` instruction guarantees that either of the two cases can only occur if all instructions prior to the `sync` instruction k have already left the pipeline in cycle t. In particular, the last store instruction $l < k$ has already left the pipeline in cycle t. Since l is a memory instruction, it cannot have entered any execution unit but the memory unit

and since instruction l has already left the pipeline, it has therefore also left the memory unit.

Hence, we find a cycle $t' < t$ where instruction l finished its memory access, i.e., $sI(mem, t') = l$ and $sI(M_I, t' + 1) = l + 1$ hold. Since the scheduling function for stage M_I is increasing on t, we know

$$sI(M_I, t) \geq sI(M_I, t' + 1) = l + 1 > l$$

which is a contradiction to $sI(M_I, t) \leq l$ and thus shows the claim. □

4.5 Correctness with interrupts

In this section, we extend VAMP correctness to also cover interrupts. As a first step, we will focus on the actual interrupt cycle and prove interrupts to be precise. Then, we will conclude overall correctness by decomposing arbitrary computations into chunks with one final interrupt each.

4.5.1 Precise interrupts

In this section, we derive correctness with interrupts from correctness without in the absence of interrupts and prove the actual interrupt cycle to preserve correctness with interrupts.

Lemma 4.5.1 *VAMP correctness with interrupts holds up to the cycle of the first interrupt, i.e.,* $\forall t \in \mathbb{N}$*:*

$$(synced\_code\_i? \wedge \forall t' \in \mathbb{Z}_t : \neg JISR(c_I^{t'})) \implies corr?(t) \wedge corr\_i?(t)$$

Proof: Let $synced\_code\_i?$ hold and $\forall t' \in \mathbb{Z}_t : \neg JISR(c_I^{t'})$. Note that $synced\_code?$ then also holds by definition 4.4.8. We apply lemma 4.3.2 in order to conclude $sI(inst, t) = sI(wb, t)$. Lemma 4.4.1 also ensures $corr?(t')$ for any $t' \leq t$ and $\tilde{c}_I^t = c_I^t$ holds according to proposition 4.2.1. Since $sI(wb, t)$ is increasing on t, we know $\neg JISR(c_S^j)$ for any $j < sI(wb, t)$ and proposition 4.1.1 ensures $\tilde{c}_S^i = c_S^i$. Hence, all the register file parts of $corr\_i?(t)$ are already concluded and we can assume $t = 0 \vee JISR(c_I^{t-1})$ for the remaining items. Because of $\neg JISR(c_I^{t'})$ for any $t' < t$, this case can be reduced to $t = 0$. Since the scheduling functions for $issue$, M_I, and $inst$ are all initialized with 0, $corr\_i?(0)$ follows from $corr?(0)$ and the proof is finished. □

Hence, correctness without interrupts implies correctness with interrupts. We now focus on the cycle after an interrupt.

Lemma 4.5.2 *VAMP correctness with interrupts also holds in the cycle immediately after the first interrupt. Formally, we have* $\forall t \in \mathbb{N}$*:*

$$(synced\_code\_i? \wedge \forall t' \in \mathbb{Z}_{t-1} : \neg JISR(c_I^{t'})) \implies corr\_i?(t)$$

Note that in case of $\neg \mathit{JISR}(c_I^{t-1})$, proposition 4.5.1 is sufficient to show the above lemma. We therefore only consider $t > 0$ and $\mathit{JISR}(c_I^{t-1})$ in the remaining section.

In the VAMP implementation, interrupts are triggered during writeback. Additionally, instructions leave the pipeline in order because of the reorder buffer. Hence, preciseness of interrupts with respect to the register files is easy to achieve. However, in order to ensure preciseness of interrupts for the memory, a `store` instruction is not allowed to start its memory access until it is the oldest instruction in the CPU. This ensures that only those `store` instructions that are not flushed by interrupts begin their memory access. The stalling is achieved by comparing the tag of the instruction in the memory unit with the tag at the head of the reorder buffer and stalling stores until equality holds.

Lemma 4.5.3 *In case the first interrupt occurs, the specification memory of the instruction in the M_I stage and the instruction after the one in the writeback stage are equal, i.e., $\forall t \in \mathbb{N}$:*

$$(\mathit{synced\_code?} \wedge \forall t' \in \mathbb{Z}_t : \neg \mathit{JISR}(c_I^{t'})) \wedge \mathit{JISR}(c_I^t) \implies \tilde{c}_S^{sI(wb,t)+1}.M = \tilde{c}_S^{sI(M_I,t)}.M$$

Proof: Let t be the first interrupt cycle. We have to show $\tilde{c}_S^{sI(wb,t)+1}.M = \tilde{c}_S^{sI(M_I,t)}.M$. Note that $sI(M_I,t)$ can be both be greater and smaller than $sI(wb,t)+1$. In case it is greater, we have to show that this can be only due to load instructions since stores wait until they are the oldest instruction in the pipeline. In case it is smaller, we have to argue that all stores prior to $sI(wb,t)+1$ have actually been executed.

1. Let $sI(M_I,t) > sI(wb,t)+1$ hold. We then find an intermediate store instruction i, $sI(wb,t)+1 \leq i < sI(M_I,t)$. Since memory instructions are executed in order, store i has started its memory access in some cycle $t' \leq t$ with $sI(mem,t') = i$. Since stores only access the memory when they are the oldest instruction in the pipeline, we conclude $i = sI(wb,t')$. Writeback occurs in order, i.e., we know $sI(wb,t') \leq sI(wb,t)$. This yields $i < sI(wb,t)+1$ which is a contradiction and finishes this case of the claim.

2. Let $sI(M_I,t) < sI(wb,t)+1$ hold. We then find an intermediate store instruction i, $sI(M_I,t) \leq i < sI(wb,t)+1$. Note that since $\mathit{JISR}(c_I^t)$ holds, we conclude that the implementation without interrupts would perform a writeback in cycle t, i.e., $sI(wb,t+1) = sI(wb,t)+1$. Since writeback occurs in order, we find a cycle $t' \leq t$ such that instruction i is written back and leaves the pipeline.[2] In particular, since

[2]Note that in case of $sI(wb,t) = i$, we actually need the assumption $\mathit{JISR}(c_I^t)$ in order to conclude that instruction i is written back in cycle $t' = t$ and not later.

the store instruction i leaves the pipeline in cycle t', we find an earlier cycle $t'' < t'$ where the store actually finished its memory access, i.e., $sI(mem, t'') = i$ and $mw_I^{t'} \wedge \neg c_I^{t'}.M.dbusy$ hold which leads to $sI(M_I, t''+1) = i+1$. Since the memory stage is passed in order, i.e., $sI(M_I, t)$ is increasing on t, we can use $t \geq t''+1$ in order to conclude $sI(M_I, t) \geq sI(M_I, t''+1) = i+1$ which is a contradiction to $sI(M_I, t) \leq i$ and thus concludes the claim. □

Lemma 4.5.4 *In case of an interrupt, no write to the data memory is currently in progress, $\forall t \in \mathbb{N}$:*

$$(\forall t' \in \mathbb{Z}_t : \neg \mathit{JISR}(c_I^{t'})) \wedge \mathit{JISR}(c_I^t) \implies \neg mw_I^t$$

Proof: We assume $\mathit{JISR}(c_I^t)$ and mw_I^t and have to find a contradiction. Because of mw_I^t, we conclude that there is an instruction in the memory unit with $sI(mem, t) = sI(wb, t)$. Hence, the oldest instruction in the pipeline is in the memory unit. On the other hand, $\mathit{JISR}(c_I^t)$ can only occur if the implementation without interrupts would perform a writeback, i.e., the oldest instruction in the pipeline is actually in the reorder buffer. This is a contradiction since the oldest instruction can only be in one of these two stages. □

Lemma 4.5.5 *Let the first interrupt occur in cycle t. The memory part of $corr\_i?(t+1)$ then holds, i.e., we have $\forall t \in \mathbb{N}$:*

$$(\mathit{synced\_code\_}i? \wedge \forall t' \in \mathbb{Z}_t : \neg \mathit{JISR}(c_I^{t'})) \wedge \mathit{JISR}(c_I^t) \implies c_I^{t+1}.M = c_S^{sI(inst,t+1)}.M$$

Proof: We set $i := sI(inst, t+1) - 1$. With lemma 4.5.4, we conclude $\neg mw_I^t$ and thus, $c_I^{t+1}.M = c_I^t.M$. Proposition 4.2.1 ensures $\tilde{c}_I^t = c_I^t$. By lemma 4.4.1, we know $corr?(t)$, i.e., in particular, $c_I^t.M = \tilde{c}_S^{sI(M_I,t)}.M$. Lemma 4.5.3 additionally guarantees $\tilde{c}_S^{sI(M_I,t)}.M = \tilde{c}_S^{sI(wb,t)+1}.M$. Because of $\mathit{JISR}(c_I^t)$, we also have $i - 1 = sI(inst, t)$. Proposition 4.3.2 finally ensures $sI(inst, t) = sI(wb, t)$. Hence, it only remains to show $\tilde{c}_S^i = c_S^i$. Proposition 4.1.1 guarantees $\tilde{c}_S^{i-1} = c_S^{i-1}$. Since $\mathit{JISR}(c_S^{i-1})$ holds, we know $c_S^i.M = c_S^{i-1}.M$ by the definition of the programmer's model; additionally, we have $\tilde{c}_S^i.M = \tilde{c}_S^{i-1}.M$ since the next step function of the memory in the programmer's model with interrupts in case of an interrupt is equal to the one without interrupt. This concludes the claim. □

For the proof of lemma 4.5.2, we thus only have to show the corresponding equations for the PCs and the register files according to definition 4.3.4.

Proof: (of lemma 4.5.2) Let $\mathit{synced\_code\_}i? \wedge \forall t' \in \mathbb{Z}_{t-1} : \neg \mathit{JISR}(c_I^{t'})$ hold. We have to show $corr\_i?(t)$. In case of $t = 0$ or $\neg \mathit{JISR}(c_I^{t-1})$, lemma 4.5.1 finishes the proof. Let therefore $t > 0$ and $\mathit{JISR}(c_I^{t-1})$ hold. We set $i :=$

$sI(inst, t-1)$. Note that because of $JISR(c_I^{t-1})$, $i+1 = sI(inst, t)$ also holds and proposition 4.3.2 ensures $i = sI(wb, t-1)$.

Lemma 4.4.1 guarantees correctness without interrupts, i.e., $corr?(t-1)$, and $\tilde{c}_I^{t-1} = c_I^{t-1}$ also holds by proposition 4.2.1. In particular, this means $c_I^{t-1}.SPR[SR]{=}c_S^i.SPR[SR]$. Because of the correctness of instruction results in the reorder buffer by the Tomasulo algorithm in cycle $t-1$, we also have $CA(c_I^{t-1}) = CA(c_S^i)$ and $EData(c_I^{t-1}) = EData(c_S^i)$. We therefore trivially conclude $repeat(c_I^{t-1}) = repeat(c_S^i)$, $MCA(c_I^{t-1}) = MCA(c_S^i)$, and $JISR(c_S^i)$.

We show the different parts of $corr\_i?(t)$ according to definition 4.3.4 separately.

1. $c_I^t.M = c_S^{i+1}.M$. This is proved by lemma 4.5.5.

2. $c_I^t.PC' = c_S^{i+1}.PC'$ and $c_I^t.DPC = c_S^{i+1}.DPC$. Since $JISR(c_I^{t-1})$ and $JISR(c_S^i)$ both hold, we have $c_I^t.DPC = SISR = c_S^{i+1}.DPC$ as well as $c_I^t.PC' = SISR + 4 = c_S^{i+1}.PC'$ and this part is finished.

3. $c_I^t.GPR[x].data = c_S^{i+1}.GPR[x]$ and $c_I^t.FPR[x].data = c_S^{i+1}.FPR[x]$. We know $repeat(c_I^{t-1}) = repeat(c_S^i)$. In case of $repeat(c_S^i)$, both implementation and specification leave the register files unchanged and this case is finished. Let us therefore assume $\neg repeat(c_S^i)$. However, we then have $c_I^t.GPR[x].data = c_I^{t-1\prime}.GPR[x].data$ and $c_S^{i+1}.GPR = c_S^i{}'.GPR$ and the correctness of the primed version has already been established during the induction step for the register file without interrupts which we omitted in this thesis. We can apply this induction step here since all its assumptions are fulfilled, in particular the Tomasulo invariant at cycle $t-1$. The same holds true for the *FPR*.

4. $c_I^t.SPR[x].data = c_S^{i+1}.SPR[x]$. For the special purpose register file, we actually have to show more than for the other two register files since several values are saved in special registers in case of an interrupt. According to equation (4.12), the special registers written in this way are *SR*, *EData*, *ECA*, *ESR*, *EPC*, and *EDPC*. Note that for all other registers, the arguments are identical to the ones for the other register files and are therefore omitted in this case.

   (a) *SR*. The status register *SR* is just forced to the value 0^{32} in both implementation and specification which makes this register an easy proof goal.

   (b) *ECA* and *EData*. The correctness of *ECA* and *EData* is straightforward since $MCA(c_I^{t-1})$ and $EData(c_I^{t-1})$, respectively, are written into these two registers in the implementation and the correctness of these two parts is already established by the above arguments.

(c) *ESR*. By expanding the definition of $c_S^{i+1}.SPR[ESR]$, we only need to prove

$$c_I^t.SPR[ESR] = repeat(c_S^i)?\, c_S^i.SPR[0] : c_S^i{}'.SPR[0]$$

In both cases, the above reasoning about the other register files can be applied and concludes this case.

(d) *EDPC* and *EPC*. By the same arguments as for the correctness of the destination register index in the reorder buffer, we know during instruction writeback that the PCs in the reorder buffer still hold the values written during instruction issue in some cycle $k < t-1$ with $sI(issue, k) = i$. Because $corr?(k)$ holds, these PCs are in particular correct, i.e., $c_I^k.PC' = c_S^i.PC'$, $c_I^{k\prime}.PC' = c_S^i{}'.PC'$, and accordingly for *DPC*. Since $repeat(c_I^{t-1})$ is correct, the correctness of both exception PCs is proved. □

Thus, we have correctness with interrupts for the VAMP up to the cycle after the first interrupt. The following lemma finally states that in the cycle after an interrupt, the VAMP is empty.

Proposition 4.5.6 *Initially and in the cycle after an interrupt, the VAMP in an initial state according to definition 4.2.2. Formally, we have* $\forall t \in \mathbb{N}$*:*

$$(t = 0 \vee JISR(c_I^{t-1})) \implies init?(c_I^t)$$

The proof that on an interrupt in the VAMP, all reservation stations, execution units, producer registers, the reorder buffer, and the decode stage are emptied and all register are set to be valid in the producer tables as required by $init?(c^t)$ is straightforward. We now basically have everything in place in order to conclude overall VAMP correctness.

4.5.2 Overall correctness

In order to extend the correctness with interrupts on interrupt-free sequences to arbitrary VAMP computations, we need the $[c_{init}]$ notation introduced before. We start with the central lemma for correctness with interrupts that basically states that correctness in some cycle $t'+1$ can be reduced to correctness in the cycle of the last interrupt before t'. We will then use this lemma in the induction step of the final theorem.

Lemma 4.5.7 *If we have a cycle* t *after an interrupt with* $corr\_i?(t)$ *and an interrupt-free interval until some cycle* $t' \geq t$*, overall correctness with interrupts also holds in cycle* $t'+1$*. Formally, we have* $\forall t \in \mathbb{N}$*:*

$$synced\_code\_i? \wedge corr\_i?(t) \wedge (t = 0 \vee JISR(c_I^{t-1})) \implies$$
$$\forall t' \in \mathbb{Z}_{\geq t} : (\forall t'' \in [t : t'[: \neg JISR(c_I^{t''})) \implies corr\_i?(t'+1)$$

Proof: Let $synced\_code\_i? \wedge corr\_i?(t) \wedge (t = 0 \vee JISR(c_I^{t-1}))$ hold. Note that $init?(c_I^t)$ follows directly form proposition 4.5.6. Equation (4.25) also ensures $spec\_conf(c_I^t) = c_S^{sI(inst,t)}$. We apply decomposition in order to conclude $c_I^{t'+1} = c_I[c_I^t]^{t'-t+1}$ and

$$sI(inst, t'+1) = sI(inst, t) + sI[c_I^t](inst, t'-t+1).$$

Therefore, we also have $c_S^{sI(inst,t'+1)} = c_S[c_I^t]^{sI[c_I^t](inst,t'-t+1)}$. This means that $corr\_i?[c_I^t](t'-t+1) = corr\_i?(t'+1)$ holds and thus, we only have to conclude $corr\_i?[c_I^t](t'-t+1)$. Since $\forall t'' \in [t : t'[: \neg JISR(c_I^{t''})$ holds and is equal to $\forall t'' \in \mathbb{Z}_{t'-t} : \neg JISR(c_I[c_I^t]^{t''})$, lemma 4.5.2 finishes the proof. □

Theorem 4.5.8 *Let the assembly code fulfill the sync condition. The VAMP is then correct.*

$$synced\_code\_i? \implies corr\_i?(t)$$

Proof: Let $synced\_code\_i?$ hold. We show $corr\_i?(t)$ by induction on t.
Induction base ($\mathbf{t = 0}$): The induction base is trivial since $corr\_i?(0)$ holds by definition.
Induction step ($\mathbf{t \to t+1}$): We have to show $corr\_i?(t+1)$. We split cases on whether an interrupt occurred prior to cycle t or not.

1. Let $\neg c_I.\exists^{last}_{JISR}(t)$ hold. This means $\forall k \in \mathbb{Z}_t : \neg JISR(c_I^k)$ according to definition 1.2.8 of $\exists^{last}$. Since we have already shown $corr\_i?(0)$ in the induction base, we can apply lemma 4.5.7 to cycles 0 and t in order to conclude $corr\_i?(t+1)$ and finish this case.

2. Let $c_I.\exists^{last}_{JISR}(t)$ hold. We set $l := c_I.last_{JISR}(t)$ and by definition 1.2.8 of $last$, we also have $l+1 \leq t$, $JISR(c_I^l)$, and $\neg JISR(c_I^k)$ for any $k \in [l+1 : t[$. By induction hypothesis, $corr\_i?(l+1)$ also holds. Hence, we can apply lemma 4.5.7 to cycles $l+1$ and t and conclude $corr\_i?(t+1)$. This finally finishes the last theorem and proves the VAMP free of errors. □

This concludes correctness of the VAMP based on the 'interim' memory definition $c_I.M$. We will plug in the proof of M_I in the following section.

4.6 Implementation on an FPGA

So far, we considered computations starting in an initial state. However, hardware usually does not have a fixed state on power up; therefore, we introduce a *reset* signal. This signal initialized the memory interface, i.e., $cache\_reset := reset$, in addition to the effect of a *JISR* we described in the implementation. Hence, a *reset* initializes *both* the VAMP and its memory interface. Note that *reset* also sets the corresponding bit 0 in the exception

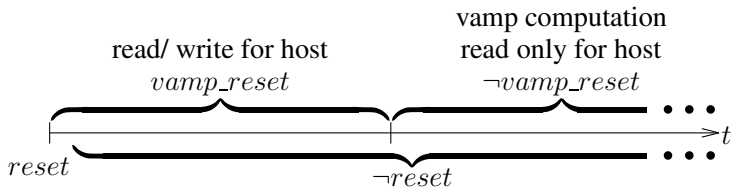

Figure 4.9: Power-up sequence of the VAMP

cause such that the interrupt handler can find out that it was called on power up rather than upon an interrupt.

Let therefore 0 be the last cycle where *reset* was active. We then have $init?(c_I^1)$ and can instantiate a correct memory interface according to definition 1.5.2 since $c_I^t.M.clear = (t = 0)$ holds. In addition, $c_I[c_I^1].M^t = M_I^{t+1}$ holds and we can plug in the memory interface and its correctness proof in order to conclude $corr\_i?[c_I^1](t)$ for any $t \in \mathbb{N}$.

For the implementation on the PCI board, we would like to have the possibility to access the VAMP memory in order to write programs into the memory and read their results back from memory. Since the cache memory interface features writeback policy, the physical memory does not hold correct data in general as we also showed in the consistency invariant $mif\_consistency$ in section 3.4.2. Therefore, it makes sense that the host actually accesses the cache memory interface of the VAMP instead of the physical memory.

We therefore designed an almost trivial extension of the cache memory interface that allows for *two* data access ports. Note that in contrast to the truly parallel instruction and data port, only one of the two data ports is served at any time. This is achieved by means of a simple fair arbiter and a multiplexer that either passes the access from data port A or data port B to the memory interface and takes care that port B is signals busy while port A is served and vice versa. Even the formal proof that for this extension, both data ports fulfill the consistency and liveness criterion of a correct memory interface is simple. Note in particular that because of the fairness of the arbiter, liveness of the data access port A is not affected by continuous accesses to data port B and vice versa, only performance suffers.

We then integrated this extended cache memory interface into the VAMP and connected its second data port to the host PC. In addition, we introduced a second reset signal $vamp\_reset$ that *only* keeps the VAMP in an initial state, while the cache memory interface is up and running. Since the VAMP initiates no requests for the cache memory interface while in reset state, the host PC has exclusive access to the VAMP memory. After the signal $vamp\_reset$ is deactivated, we only allow read accesses from the host PC. This power-up sequence is illustrated in figure 4.9.

In summary, we actually verified the following claim for the VAMP im-

plementation: Let an initial *reset* be followed by an arbitrarily long period where only *vamp_reset* is active and the host reads or writes the memory as he chooses. As soon as *vamp_reset* is deactivated, the current memory content the host sees via the definition of a memory interface becomes the initial memory content of the specification and both implementation computation and specification computation start. Naturally, the implementation is correct with respect to the specification. Note that the host can perform arbitrary read accesses on the memory interface during the computation. In particular, these reads may cause writeback in case of a dirty miss or invalidation in the instruction cache during snooping. However, the formally verified layer of the memory interface guarantees that VAMP correctness is not affected.

4.7 Related work

This section is based on [BJK+03, Sect. 1].

The formal verification of a Tomasulo schedulers with reorder buffers is by no means new [DP97,HSG99,McM98,SH98]. Exploiting symmetries, McMillan [McM98] has shown the correctness of a powerful Tomasulo scheduler with a remarkable degree of automation. Using theorem proving, Sawada and Hunt [SH98] show the correctness of an entire out-of-order processor, precise interrupts, and a store buffer for the memory unit. They also consider self-modifying code by means of a `sync` instruction. However, none of the above authors introduces the little irregularities in the Tomasulo algorithm we allow, i.e., register 0 in the *GPR* always returns 0 and the *IEEEf* implementation which break standard Tomasulo forwarding.

For a discussion of the related word for the floating point units of the VAMP, we refer to other publications in the VAMP project [Ber01, BJ01, Jac02a, Jac02b] where the actual implementation and verification of the FPUs is also described.

Brock and Hunt report the formal verification of the simple, non-pipelined FM9001 processor [BHK94] whose complexity is clearly far below the VAMP. Apart from this FM9001 processor and our VAMP project, *none* of the papers quoted above states that the verified design actually has been implemented. *All* results outside the VAMP project except [BHK94] use several simplifications and abstractions:

1. The realized instruction set is restricted: always included are the six instructions considered in [BD94]: load word, store word, jump, branch equal zero, ALU register operations, ALU immediate operations. Five typical extra instructions are trap, return from exception, move to and from special registers, and sync [SH98]. The branch equal zero instruction is generalized in [VB00] by an uninterpreted test evaluation function. Most notably the verification of machines with load/store

operations on half words and bytes has apparently not been reported. In [VB99] the authors report an attempt to handle these instructions by automatic methods which was unsuccessful due to memory overflow.

2. Sometimes, non-implementable constructs are used in the processors: e.g., Hosabettu et.al. [HSG99] use tags from an infinite set. Obviously, this is not directly implementable in real hardware.

3. The verification of pipelines or Tomasulo schedulers with *instantiated* floating point units and memory units with caches and main memory bus protocol has not been reported. Indeed, in [VB99] the authors state: "An area of future work will be to prove that the correctness of an abstract term-level model implies the correctness of the original bit-level design."

Chapter 5

Conclusion

5.1 Summary

In this thesis, we presented implementations of different caches and formally verified their correctness. These caches are parameterized over their associativity, the address width of the memory, the number of bytes in a data word, the size of cache sectors, and the width of tags. They also support write back policy by keeping a dirty bit in their directory.

As a next step, we integrated these caches into a parameterized cache memory interface. Note that the instruction and data cache in the cache memory interface have to share several parameters, i.e., address and data width as well as sector size, but they may actually differ in associativity and tag width since these parameters are not visible to the cache memory interface. We formalized a bus protocol with bursts since the two caches are connected to a physical memory accessed via a bus protocol. We formally verified that the cache memory interface with write back policy for the data cache is correct, i.e., it behaves like a dual-ported memory with arbitrary variable latency. Note that in the correctness criterion, the size of cache sectors is no longer visible, but only address and data width.

Finally, we have also described an implementation of the Tomasulo algorithm with a memory unit, a fixed point unit, and three floating point units called VAMP. The memory unit of the VAMP consists of an instantiated cache memory interface and also performs instruction fetch. The floating point units and their formal verification is described in [Jac02a]. For performance reasons, we have extended the Tomasulo algorithm by special treatment for register *IEEEf*. We have also added support for precise, maskable interrupts in the VAMP. We addressed the problem of self-modifying code by means of a `sync` instruction.

In addition to the implementation, we also presented the programmer's model of the VAMP which serves as a specification in the correctness proof. We have shown overall correctness of this VAMP implementation with re-

spect to the specification focusing on data memory accesses, instruction fetch and self-modifying code, the *IEEEf* extension of the Tomasulo algorithm, and precise interrupts. The VAMP implementation was synthesized with an automated tool and is currently running on a Xilinx FPGA on a PCI board in a host PC at our institute. The Xilinx software reports an equivalent gate count of about 1.5 million gates for the synthesized VAMP. Note that this includes 8 Kbyte of data memory for the instruction cache and 16 Kbyte for the data cache.

5.2 Discussion

We have presented the design of a complex 32-bit microprocessor and verified it against its specification. The design is based on [MP00, Jac02a, Krö01] and therefore might be considered purely academic. However, consider that we implement a *full* DLX instruction set with memory operations on bytes, halfwords, words, and doubles, as well as signed or unsigned operations on loads. The interrupt support we provide allows for masking of interrupts as well as repeat or continue interrupts. In addition, our implementation has many—from the point of verification—nasty details that break the simple symmetry so often found in academic project. Consider, e.g., the fact that register 0 in the *GPR* always return 0. We therefore have to take care that register 0 is not forwarded in our Tomasulo implementation. In addition, we have mixed single and double precision operations and the *IEEEf* extension that break the standard Tomasulo forwarding. We therefore claim that the VAMP is not a purely academic design, but actually an industrial-strength CPU, and that verification in the style of this thesis can be carried out on real world architectures.

In the VAMP project, we used a functional subset of the PVS language for modeling our hardware and verified it in PVS itself. Thus, the PVS system served as hardware development and verification system. In the remaining section which is based on [BJKL02, Sect. 6], we will discuss the pitfalls and benefits of this approach.

1. Designing combinational circuits in a functional programming language and our notion of clocked circuits is not common practice for hardware designers.

2. The support for fast simulation and visualization is common in modern development systems, but not available in PVS. In the design phase, many obvious errors can be found by simulation. The harder errors could then be found during formal verification. The theorem prover ACL2 [KM96], on the other hand, offers efficient lisp-based support for simulation. However, ACL2 cannot handle higher-order logic in contrast to PVS which would considerably complicate our proofs.

3. We support only a single clock domain. Thus, we cannot directly model a common SDRAM interface of a CPU in PVS where the SDRAM is clocked independently of the CPU. An extension of our PVS hardware model to cover multiple clock domains is possible, but we have not yet investigated this possibility.

4. Our PVS hardware model supports only a small subset of the Verilog hardware description language which is sufficient to design any combinational circuit or clocked circuit with one single clock domain. However, by designing hardware in PVS, we disallow any "dirty" design tricks employed in common HDLs in order to achieve a maximum optimization of the design. Therefore, it may not be possible to design hardware as thoroughly optimized for speed as the latest Pentium generation, for instance. However, it is not our project goal to compete with modern microprocessors in performance, but to offer formally verified correctness guarantees for microprocessors in safety-critical devices. Many of these safety-critical devices do not need a clock frequency of more than 400 MHz which could be achieved by our approach. We see a considerable market for formally verified microprocessor of comparably modest performance, e.g., in medical devices, nuclear reactors, and military applications.

5. A considerable part of the verification effort is needed for very low-level circuits for which appropriate automatic methods are available, e.g., equivalence-checking. One could save a great deal of time by automatically verifying small sub-circuits, and restrict interactive proof development to the composition of such sub-circuits to larger circuits which are too large for automatic verification. However, these automatic methods are not available in PVS.

There are publicly available tools supporting some of these features, but none integrates all features needed for an integrated development & verification system. There are such tools in industry, e.g. Intel's Forte system [OZGS99], but these tools are not publicly available, they are not even sold. However, there are many advantages to our approach of designing and verifying hardware.

1. The use of high-level constructs such as recursion and λ-expressions allows for the concise description of structured hardware.

2. The description of hardware in PVS enables the formal verification of the hardware descriptions against some formal specification.

3. The PVS system offers support for both theorem proving as well as model checking. Thus, we can exploit both techniques in our proofs

without a tedious and error-prone translation between two different verification systems.

4. The verification can exploit the structured and modular description of the hardware; one can verify general purpose circuits for arbitrary bit widths, and use the correctness results in the verification of larger and larger circuits. In this way, it is possible to design, verify and implement hardware of almost arbitrary complexity which we actually did by verifying the whole VAMP.

The latter points are particularly important, as the design of complex hardware systems is very error prone, and verification is therefore an increasingly important part of the development cycle.

Hardware is specified and verified in PVS on the gate level. In order to obtain real hardware, we have developed the `pvs2hdl` tool to automatically translate the PVS hardware descriptions to Verilog. Several other tools (synthesizer, place & route tools, etc.) then transform Verilog to real hardware. Each of the steps involved is not formally verified and could introduce new errors into the design. In fact, even the PVS proof checker could have bugs which hide errors in the "verified" PVS hardware description.

However, there is a great benefit in having verified the PVS gate-level description of the hardware: the design is free of *logical* errors (if we have not been trapped by bugs in PVS). Nowhere an *and*-gate is used where an *or*-gate would have been correct, no adder is too small in size,... Although each of the tools mentioned above could introduce new errors, the confidence in the logical correctness of the gate-level greatly improves the confidence in the correctness of the ultimate hardware. The infamous Pentium bug, for example, was a logical bug [Pra95] which would have been discovered in our verification. There are approaches to verified synthesis tools [AL95, ML01]. However, the formal verification of real-size synthesis tools is far beyond the capabilities of current software verification techniques.

5.3 Future work

Since all the proofs presented in this thesis were developed interactively in PVS, one direction of future work is obvious: We need a theorem proving system with integrated powerful automated tools. With such a complete system, we believe a speedup of the verification by the factor of two or more could be achieved since many of the proof goals are tedious and of sufficiently small complexity in order to allow for automated proofs.

For the memory interface, one could imagine a memory hierarchy, i.e., the instruction and data caches are connected to a unified second-level cache which in turn accesses the main memory. However, this does not involve any new arguments in the proof: For the first level caches, only the data

transmission protocol between first and second level caches has to be adapted from the bus protocol in chapter 3 to a version more suited for efficient on-chip communication. The arguments for the second level itself are just a very much simplified version of the proof in chapter 3. As a matter of course, this verification is again performed in layers, i.e., the second level cache is verified stand-alone and then its consistency and liveness is integrated into the proof for the split first level.

A further extension of our cache memory interface stems from the fact that upon integrating it into a memory unit, any access on the data port is blocking, i.e., in case of a miss which might entail the writeback of a dirty line and the fetch of new data, no other access can be processed until the previous access terminates even if the new access produces a hit and could therefore be processed in one cycle. However, the central consistency invariant $mif\_consistency$ we showed in chapter 3 for our cache memory interface ensures correct data in the data cache on a hit. Thus, if we replace the data RAM of the data cache with a dual-ported data RAM which allows for a second read access on address adr' and copy the computation of $dout$ and hit for this second read port, the consistency invariant of the memory interface ensures correct data for *both* these access ports on a hit practically without any effort.

Hence, we just connect the additional input adr' and the two additional outputs hit' and $dout'$ directly to the memory unit. Now, any read in the memory unit that produces a hit' on the address port adr' can overtake any pending access on the regular access port. On the other hand, any write or miss still would have to wait for the first instruction to terminate its memory access. In addition, if one would like to keep LRU replacement strategy, one would have to integrate such an out-of-order access into the history computation. The proof effort involved in this extension without the LRU integration is comparatively modest. The hardware cost, on the other hand, rises drastically by replacing the data memory of a cache with a dual-ported version. One would have to investigate the performance boost of this implementation by simulation in order to decide whether it pays off.

The memory unit of the VAMP is very simple. It can only hold one instruction at a time. One could easily imagine a more complex memory unit which can process several instruction simultaneously, features store buffers, and allows reordering of instructions. However, this would require additional effort in ensuring the preciseness of interrupts.

Currently, we have separate liveness proofs of the Tomasulo scheduler and all its execution units. Clearly, it would be desirable to formally close the remaining gap and show liveness for the VAMP.

The version of the VAMP we presented here has no hardware support for address translation. However, common operating systems require paging and address translation on the hardware in order to give each user program its own virtual memory. Jakov Dalinger currently works on adding a sim-

ple memory management hardware to the VAMP [Dal04]. Apart from the integration of interrupts, his work is finished. However, in addition to the hardware, the handlers of the page fault interrupts in software are important since memory management spans both hardware and system software. Mark Hillebrand gives correctness proofs of memory management spanning both these layers [Hil05].

One important result of this thesis lies just in the programmer's model of the VAMP since it allows for the verification of arbitrary assembler code. Recent work at our institute also developed a layer on top of the programmer's model that eases code verification and closed the gap between these two layers [Par04, Shm04].

However, there are still several layers on top of assembly code before we reach the application layer and can claim correctness of a typical system. As a part of the Verisoft[1] project funded by the German federal government, we are therefore carrying the hierarchical proof approach several steps further: We add a formally verified compiler [Pet04] of a subset of the C language [Lei04] and a simple operating system [Gar04, Bog04]; at the top, we run an email client, some signature software, and TCP/IP-protocol support as applications. The ambitious goal is to seamlessly integrate all these different layers into one single correctness statement that signing and sending or receiving and checking of the signature, respectively, are correct *on the VAMP architecture*. Therefore, several additional layers with appropriate black boxes between instruction set architecture level and application software are in the process of being developed. We intend to achieve these goals by mid 2007.

In order to complete the circle, let us once again return to HAL 9000 from the introduction. In this thesis, we covered the correctness of a complex 32-bit microprocessor. The Verisoft project builds an operating system with an email client of top of the VAMP and formally verifies overall correctness. Although an email client is admittedly not even close to the complexity of HAL, it decreases the remaining gap to the formal verification of HAL drastically. Only a comparatively small amount of steps remain which could be taken in the next decade or two. We will close this thesis with the 'dying' song of HAL 9000 from [Kub68] in full view of the fact that it would not have taken place with formal verification.

'Daisy, Daisy, give me your answer do.
I'm half crazy all for the love of you.
It won't be a stylish marriage,
I can't afford a carriage.
But you'll look sweet
upon the seat
of a bicycle built for two.'

[1] http://www.verisoft.de

Appendix A

VAMP instruction set

The VAMP instruction set is taken from [Krö01] with minimal modifications.

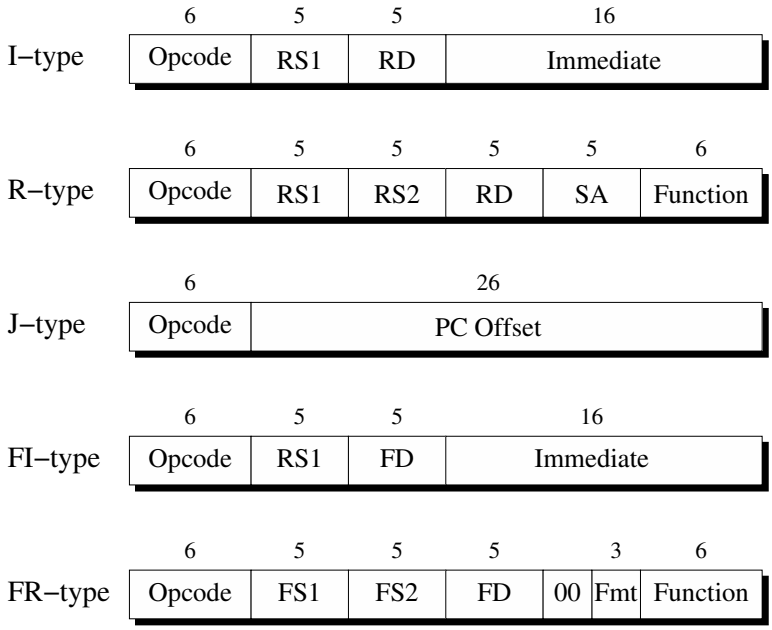

Figure A.1: Instruction formats of the VAMP

$IR[31:26]$	Mnem.	d	Effect
Memory operations, $ea = RS1 + imm$			
100000	`lb`	1	$RD = sext(M[ea + d - 1 : ea])$
100001	`lh`	2	$RD = sext(M[ea + d - 1 : ea])$
100011	`lw`	4	$RD = M[ea + d - 1 : ea]$
100100	`lbu`	1	$RD = 0^{24}M[ea + d - 1 : ea]$
100101	`lhu`	2	$RD = 0^{16}M[ea + d - 1 : ea]$
101000	`sb`	1	$M[ea + d - 1 : ea] = RD[7 : 0]$
101001	`sh`	2	$M[ea + d - 1 : ea] = RD[15 : 0]$
101011	`sw`	4	$M[ea + d - 1 : ea] = RD$
Arithmetic, logical operation			
001000	`addi`		$RD = RS1 + imm$
001001	`addiu`		$RD = RS1 + imm$ (no overflow)
001010	`subi`		$RD = RS1 - imm$
001011	`subiu`		$RD = RS1 - imm$ (no overflow)
001100	`andi`		$RD = RS1 \wedge imm$
001101	`ori`		$RD = RS1 \vee imm$
001110	`xori`		$RD = RS1 \oplus imm$
001111	`lhgi`		$RD = imm0^{16}$
Test and set operations			
011000	`clri`		$RD = 0^{32}$
011001	`sgri`		$RD = 0^{31}(RS1 > imm)$
011010	`seqi`		$RD = 0^{31}(RS1 = imm)$
011011	`sgei`		$RD = 0^{31}(RS1 \geq imm)$
011100	`slsi`		$RD = 0^{31}(RS1 < imm)$
011101	`snei`		$RD = 0^{31}(RS1 \neq imm)$
011110	`slei`		$RD = 0^{31}(RS1 \leq imm)$
011111	`seti`		$RD = 0^{31}1$
Control operation			
000100	`beqz`		$PC' = PC' + 4 + (RS1 = 0?\, imm : 0)$
000101	`bnez`		$PC' = PC' + 4 + (RS1 \neq 0?\, imm : 0)$
000110	`jr`		$PC' = RS1$
000111	`jalr`		$R31 = PC' + 4; PC' = RS1$

Table A.1: I-type instruction layout

$IR[5:0]$	Mnem.	Effect
Shift operations		
000000	`slli`	$RD = RS1 \ll SA$
000001	`slai`	$RD = RS1 \ll SA$ (arith.)
000010	`srli`	$RD = RS1 \gg SA$
000011	`srai`	$RD = RS1 \gg SA$ (arith.)
000100	`sll`	$RD = RS1 \ll RS2[4:0]$
000101	`sla`	$RD = RS1 \ll RS2[4:0]$ (arith.)
000110	`srl`	$RD = RS1 \gg RS2[4:0]$
000111	`sra`	$RD = RS1 \gg RS2[4:0]$ (arith.)
Data transfer		
010000	`movs2i`	$GPR[RD] = SPR[SA]$
010001	`movi2s`	$SPR[SA] = GPR[RS1]$
Arithmetic and logical operations		
100000	`add`	$RD = RS1 + RS2$
100001	`addu`	$RD = RS1 + RS2$ (no overfl.)
100010	`sub`	$RD = RS1 - RS2$
100011	`subu`	$RD = RS1 - RS2$ (no overfl.)
100100	`and`	$RD = RS1 \wedge RS2$
100101	`or`	$RD = RS1 \vee RS2$
100110	`xor`	$RD = RS1 \oplus RS2$
100111	`lhg`	$RD = RS2[15:0]0^{16}$
Test and set operations		
101000	`clr`	$RD = 0^{32}$
101001	`sgr`	$RD = 0^{31}(RS1 > RS2)$
101010	`seq`	$RD = 0^{31}(RS1 = RS2)$
101011	`sge`	$RD = 0^{31}(RS1 \geq RS2)$
101100	`sls`	$RD = 0^{31}(RS1 < RS2)$
101101	`sne`	$RD = 0^{31}(RS1 \neq RS2)$
101110	`sle`	$RD = 0^{31}(RS1 \leq RS2)$
101111	`set`	$RD = 0^{31}1$

Table A.2: R-type instruction layout
Note that $IR[31:26] = 0^6$ holds for all instructions in this table and that we identify a boolean value of *true* with 1 and *false* with 0.

$IR[31:26]$	Mnem.	Effect
000010	`j`	$PC' = PC' + 4 + imm$
000011	`jal`	$GPR[31] = PC' + 4$; $PC' = PC' + 4 + imm$
111110	`trap`	$trap = 1$; $EData = imm$
111111	`rfe`	$SR = ESR$; $PC' = EPC$; $DPC = EDPC$

Table A.3: J-type instruction layout

$IR[31:26]$	Mnem.	d	Effect
Memory operations, $ea = RS1 + imm$			
110001	`load.s`	4	$FD[31:0] = M[ea+d-1:ea]$
110101	`load.d`	8	$FD[63:0] = M[ea+d-1:ea]$
111001	`store.s`	4	$M[ea+d-1:ea] = FD[31:0]$
111101	`store.d`	8	$M[ea+d-1:ea] = FD[63:0]$
Control operations			
000110	`fbeqz`		$PC' = PC' + 4 + (FCC = 0 ? imm : 0)$
000111	`fbnez`		$PC' = PC' + 4 + (FCC \neq 0 ? imm : 0)$

Table A.4: FI-type instruction layout

$IR[5:0]$	$IR[8:6]$	Mnem.	Effect
Arithmetic and compare operations			
000000		`fadd`	$FD = FS1 + FS2$
000001		`fsub`	$FD = FS1 - FS2$
000010		`fmul`	$FD = FS1 * FS2$
000011		`fdiv`	$FD = FS1 \div FS2$
000100		`fneg`	$FD = -FS1$
000101		`fabs`	$FD = abs(FS1)$
000110		`fsqt`	$FD = sqrt(FS1)$
000111		`frem`	$FD = rem(FS1, FS2)$
$11c[3:0]$		`fc.cond`	$FCC = (FS1\, c\, FS2)$
Data transfer			
001000	000	`fmov.s`	$FD[31:0] = FS1[31:0]$
001000	001	`fmov.d`	$FD[63:0] = FS1[63:0]$
001001		`mf2i`	$GPR[FD] = FPR[FS1][31:0]$
001010		`mi2f`	$FPR[FD][31:0] = GPR[FS2]$
Conversion			
100000	001	`cvt.s.d`	$FD = cvt(FS1, s, d)$
100000	100	`cvt.s.i`	$FD = cvt(FS1, s, i)$
100001	000	`cvt.d.s`	$FD = cvt(FS1, d, s)$
100001	100	`cvt.d.i`	$FD = cvt(FS1, d, i)$
100100	000	`cvt.i.s`	$FD = cvt(FS1, i, s)$
100100	001	`cvt.i.d`	$FD = cvt(FS1, i, d)$

Table A.5: FR-type instruction layout
Note that $IR[31:26] = 010001$ holds for all instructions in this table.

Condition			Relations				Invalid
Code	Mnemonic		Greater	Less	Equal	Unordered	if
c	True	False	$>$	$<$	$=$	?	unordered
0000	F	T	0	0	0	0	
0001	UN	OR	0	0	0	1	
0010	EQ	NEQ	0	0	1	0	
0011	UEQ	OGL	0	0	1	1	
0100	OLT	UGE	0	1	0	0	No
0101	ULT	OGE	0	1	0	1	
0110	OLE	UGT	0	1	1	0	
0111	ULE	OGT	0	1	1	1	
1000	SF	ST	0	0	0	0	
1001	NGLE	GLE	0	0	0	1	
1010	SEQ	SNE	0	0	1	0	
1011	NGL	GL	0	0	1	1	
1100	LT	NLT	0	1	0	0	Yes
1101	NGE	GE	0	1	0	1	
1110	LE	NLE	0	1	1	0	
1111	NGT	GT	0	1	1	1	

Table A.6: Floating-point relational operators for the `fc` instruction

Appendix B

Lemmas in PVS

In this chapter we list for each lemma in this thesis the corresponding name in PVS. Lemmas in this thesis are identified by their unique *number*. In PVS, they are identifies by both a *context* and a *name*. We first give an overview over the different PVS contexts and then a separate table for the lemmas of each PVS context. Note that PVS contexts equal directories in the tree of the VAMP sources at our project homepage.

PVS context	Contents
`basics`	basic circuits (cf. section 1.4)
`predicates`	definition of *last* and *next* as well as basic lemmas (cf. definition 1.2.8 and proposition 1.2.9)
`ram`	definition of RAM (cf. definition 1.2.6)
`cache`	direct mapped cache (cf. section 2.3)
`history`	LRU history updates (cf. section 2.4.1)
`sa_cache`	set-associative cache (cf. section 2.4)
`fa_cache`	fully associative cache (cf. section 2.5)
`memory`	bus protocol (cf. section 3.1)
`memory_interface`	correct memory interface (cf. definition 1.5.2)
`pipe_control`	cache memory interface (cf. chapter 3)
`dlxif`	programmer's model by Kröning [Krö01] with our *IEEEf* extension and `sync` criterion (cf. section 4.1 and definition 4.4.8)
`tomasulo`	Tomasulo algorithm by Kröning [Krö01] with our extension for *IEEEf* and alternative correctness criteria of FPU (cf. section 4.4.1 and 4.3.3)
`dlxtom`	VAMP (cf. chapter 4)

Table B.1: Overview of PVS contexts

Name in PVS	Number
mux_tree_rec_correct	1.4.3
mux_tree_unary_select_correct	1.4.4
pp_lemma	1.4.6
encf_lemma3	1.4.8
encf_lemma1	1.4.9
encf_lemma2	1.4.10

Table B.2: Lemmas in PVS context `basics`

Name in PVS	Number
dm_cache_create_hit	2.3.1
dm_cache_create_hit_implies_vw_and_valid	2.3.2
exists_last_write_before_hit	2.3.3
byte_change_is_cdwb_ind	2.3.4
dm_cache_consistency_induction_step_helper	2.3.5
dm_cache_no_tag_change_after_same_sect_hit	2.3.6
dm_cache_no_tag_change_after_hit	2.3.7
dm_cache_no_valid_change_without_vw_ind	2.3.8
cdwb_is_hit_or_not_valid	2.3.9
hit_same_line_cdwb_is_same_tag	2.3.10
dm_cache_consistency_induction	2.3.11
dm_cache_no_linefill_after_last_write	2.3.12
dm_cache_last_write_is_hit_or_linefill	2.3.13
dm_cache_no_clear_after_last_write	2.3.14
dm_cache_vw_valid_is_miss	2.3.15
dm_cache_continuous_hit_after_last_write	2.3.16

Table B.3: Lemmas in PVS context `cache`

Name in PVS	Number
pp_input_correct	2.4.14
hsel_input_correct	2.4.15
hsel_a_mux_correct	2.4.16
hsel_b_mux_correct	2.4.17
next_history_meets_spec	2.4.18

Table B.4: Lemmas in PVS context `history`

Name in PVS	Number
no_change_without_cache_rd	2.4.2
sa_cache_way_reg_half_correct	2.4.3
sa_cache_no_change_without_wayreg_ind	2.4.4
dm_cache_last_hit_is_last_hit	2.4.5
dm_cache_good_linefill_input	2.4.6
sa_cache_no_way_change_on_different_line_ind	2.4.8
sa_cache_hit_change_on_different_tag_is_miss	2.4.9
sa_cache_hit_vector_smaller_than_last_way_reg	2.4.10
sa_cache_hit_stays_unary_on_different_sect	2.4.11
sa_cache_hit_stays_unary_on_same_sect	2.4.12
sa_cache_hit_stays_unary	2.4.13
history_forwarding_correct	2.4.19
sa_cache_hit_is_one_bit	2.4.20
sa_cache_way_reg_correct	2.4.21
sa_cache_way_reg_is_singleton	2.4.22
sa_cache_hit_is_last_wayreg	2.4.23
sa_cache_hit_equal_after_last_write	2.4.24
sa_cache_no_byte_write_after_last_write_to_hit_way	2.4.25
sa_cache_last_write_to_hit_way	2.4.26
sa_cache_data_consistency	2.4.27

Table B.5: Lemmas in PVS context `sa_cache`

Name in PVS	Number
sa_cache_dout_is_fa_cache_dout	2.5.2
next_fa_cache_is_next_sa_cache_on_singleton_way_reg	2.5.3
sa_cache_input_last_byte_write_is_last_byte_write	2.5.4
fa_cache_is_sa_cache	2.5.5
fa_cache_io_consistency	2.5.6

Table B.6: Lemmas in PVS context `fa_cache`

Name in PVS	Number
last_icache_rd_is_cache_rd	3.2.1
last_dcache_rd_is_cache_rd	3.2.2
no_simultaneous_linefill	3.4.1
linewrite_linefill_helper	3.4.2
memory_input_is_good_helper	3.4.3
last_memory_access_state	3.4.4
imr_on_not_linv	3.4.5
dcache_output_stall	3.4.6
dcache_output_stall2	3.4.7
icache_address_is_same_sect_on_linv	3.4.8
icache_input_is_good	3.4.9
dcache_input_is_good	3.4.10
complete_dline_fill	3.4.11
complete_dline_write	3.4.12
complete_iline_fill	3.4.13
dcache_input_is_good_linefill_helper	3.4.14
dcache_input_is_good_linewrite_helper	3.4.15
icache_input_is_good_linefill_helper	3.4.16
linefill_write_is_memory_read	3.4.17
memory_write_is_cache_read	3.4.18
ilinefill_write_is_memory_read_or_dcache_read	3.4.19
dcache_input_is_good_linefill	3.4.20
icache_input_is_good_linefill	3.4.21
no_dcache_write_on_iline_fill	3.4.22
no_dcache_write_after_last_iline_fill	3.4.23
mem_and_mem_spec_equal_before_non_dirty_miss_iline_fill	3.4.24
cache_subset_of_memory_induction_step_1	3.4.25
cache_subset_of_memory_induction_step_2	3.4.26
dirty_miss_before_dirty_or_hit_loss	3.4.27
cache_subset_of_memory	3.4.28

Table B.7: Lemmas in PVS context `pipe_control`

Name in PVS	Number
exists_rs_cycle	3.4.29
fwd_word_correct	3.4.30
pipe_control_consistent_from_cache_rd	3.4.31
icache_consistent	3.4.32
cache_rd_after_not_mbusy	3.4.33
cache_liveness_sufficient_from_last_cache_rd_helper	3.4.34
cache_liveness_sufficient_from_last_cache_rd	3.4.35
icache_reaches_cache_rd	3.4.36
pipe_control_live_from_cache_rd	3.4.37
dcache_snoop_live_from_cache_rd	3.4.38
pipe_control_live	3.4.39
icache_live	3.4.40
pipe_control_correct	3.4.41

Table B.8: Lemmas in PVS context `pipe_control` (continued)

Name in PVS	Number
vamp_conf_without_interrupt_equal	4.2.1
sI_inst_helper	4.3.2
vamp_correct_with_interrupt_extended	4.4.1
mem_conf_const_helper	4.4.5
mem_result_correct	4.4.6
vamp_induction_step_mem_conf	4.4.7
vamp_correct_fetch_PC	4.4.9
vamp_induction_step_S1	4.4.10
correctness_without_interrupt_step	4.5.1
correctness_with_interrupt_step	4.5.2
mem_commit_equal_mem	4.5.3
vamp_correct_JISR_step_mem	4.5.4
vamp_correct_JISR_step_mem	4.5.5
VAMP_initial_after_interrupts	4.5.6
sI_inst_helper	4.5.7
sI_inst_correct	4.5.8

Table B.9: Lemmas in PVS context `dlxtom`

Bibliography

[AL95] M. Aagaard and M. Leeser. Verifying a logic-synthesis algorithm and implementation: A case study in software verification. *IEEE Trans. on Software Engeneering*, 21(10), Oct 1995.

[Amj04] Hassan Amjad. Model checking the AMBA protocol in HOL. Technical Report 602, University of Cambridge, Computer Laboratory, 2004.

[BBJ+02] Christoph Berg, Sven Beyer, Christian Jacobi, Daniel Kröning, and Dirk Leinenbach. Formal verification of the VAMP microprocessor (project status). In *Symposium on the Effectiveness of Logic in Computer Science (ELICS02)*, number MPI-I-2002-2-007, pages 31–36. Max-Planck-Institut für Informatik, March 2002.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In *CAV 94*, volume 818, pages 68–80. Springer-Verlag, 1994.

[Ber01] Christoph Berg. Formal verification of an IEEE floating point adder. Master's thesis, Saarland University, Saarbrücken, Germany, May 2001.

[BHK94] B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic Inc., 1994.

[BJ01] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. In *CHARME 2001*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[BJK01] Christoph Berg, Christian Jacobi, and Daniel Kröning. Formal verification of a basic circuits library. In *Proc. 19th IASTED International Conference on Applied Informatics, Innsbruck (AI'2001)*, pages 252–255. ACTA Press, 2001.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP. In *CHARME 2003*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK+05] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together - formal verification of the VAMP. *International Journal of Software Tools for Technology Transfer, Special Issue on 'Recent Advances in Harware Verification' (to appear)*, 2005.

[BJKL02] Sven Beyer, Christian Jacobi, Daniel Kröning, and Dirk Leinenbach. Correct hardware by synthesis from PVS, 2002. Internal Report, available at http://busserver.cs.uni-sb.de/publikationen/BJKL02.pdf.

[BMSG96] Ricky Butler, Paul Miner, Mandayam Srivas, and Dave Greve. A bitvectors library for PVS. Technical Report TM-110274, NASA Langley Research Center, 1996.

[Bog04] Sebastian Bogan. *Formal Verification of a Simple Operating System (Draft)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[Cla90] Arthur C. Clarke. *2001: A Space Odyssey*. Orbit, special edition, 1990.

[Dal04] Jakov Dalinger. *Formal Verification of Memory Management Units (Draft)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[DP97] Werner Damm and Amir Pnueli. Verifying out-of-order executions. In *Charme IFIP WG10.5*, pages 23–47, Montreal, Canada, 1997. Chapman & Hall.

[Gar04] Mauro Gargano. *Formal Verification of Microkernels (Draft)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss (under appraisal)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[HSG99] Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In *Computer-Aided Verification, CAV '99*, volume 1633, pages 47–59. Springer-Verlag, 1999.

[Ins85] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985.

[Jac02a] Christian Jacobi. *Formal Verification of a fully IEEE-compliant Floating-Point Unit.* PhD thesis, Saarland University, Saarbrücken, Germany, 2002.

[Jac02b] Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. In *Computer Aided Verification, 14th International Conference, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[JK00] Christian Jacobi and Daniel Kröning. Proving the correctness of a complete microprocessor. In *Proc. of the 30. Jahrestagung der Gesellschaft für Informatik*. Springer, 2000.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*. National Institute of Standards and Technology, 1996.

[Krö99] Daniel Kröning. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, Saarland University, Saarbrücken, Germany, 1999.

[Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors.* PhD thesis, Saarland University, Saarbrücken, Germany, 2001.

[Kub68] Stanley Kubrick. *2001: A Space Odyssey.* AOL Time Warner Company, Motion Picture, 1968.

[Lei02] Dirk Leinenbach. Implementierung eines maschinell verifizierten Prozessors. Master's thesis, Saarland University, Saarbrücken, Germany, 2002.

[Lei04] Dirk Leinenbach. *Formal Verification of a Functional Compiler of a C-like Language (Draft).* PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[LMW86] Jacques Loeckx, Kurt Mehlhorn, and Reinhard Wilhelm. *Grundlagen der Programmiersprachen.* Teubner, Stuttgart, 1986.

[McM98] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV 98*, volume 1427. Springer, June 1998.

[McM01] K.L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME 2001*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[Mey02] Carsten Meyer. Entwicklung einer Laufzeitumgebung für den VAMP-Prozessor. Master's thesis, Saarland University, Saarbrücken, Germany, 2002.

[MK00] Silvia M. Müller and Daniel Kröning. The impact of write-back on the cache performance. In *Proc. of the IASTED International Conference on Applied Informatics, Innsbruck (AI 2000)*, pages 213–217. ACTA Press, 2000.

[ML01] Steve McKeever and Wayne Luk. Towards provably-correct hardware compilation tools based on pass separation techniques. In *Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*. Springer, 2001.

[MP95] Silvia M. Müller and Wolfgang J. Paul. *The Complexity of Simple Computer Architectures*. LNCS. Springer, 1995.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *CADE 11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.

[OZGS99] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999.

[Par04] Oleg Parshin. Formal simulation of machine instructions with interrupts by assembler instructions. Master's thesis, Saarland University, Saarbrücken, Germany, 2004.

[PD96] S. Park and D.L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padula, Italy, 1996.

[Pet04] Elena Petrova. *Formal Verification of Compilers on the Source Code Level (Draft)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[Pra95] V. R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.

[RMK03] Abhik Roychoudhury, Tulika Mitra, and S.R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. In *Conference on Design Automation and Test in Europe (DATE)*, 2003.

[SAR99] Xiaowei Shen, Arvind, and Larry Rudolph. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. In *International Conference on Supercomputing*, 1999.

[SB03] Julien Schmaltz and Dominique Borrione. Validation of a parameterized bus architecture using ACL2. In *4th International Workshop on the ACL2 Theorem Prover and Its Applications*, July 2003.

[SH98] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV 98*, volume 1427 of *LNCS*. Springer, 1998.

[Shm04] Gennady Shmonin. Standard techniques for verification of straight-line programs and loops in assembler. Master's thesis, Saarland University, Saarbrücken, Germany, 2004.

[SSA01] Joseph Stoy, Xiaowei Shen, and Arvind. Proofs of correctness of cache-coherence protocols. In *FME*, volume 2021 of *LNCS*. Springer, 2001.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research & Development*, volume 11 (1), pages 25–33. IBM, 1967.

[VB99] Miroslav N. Velev and Randal E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *CHARME*, volume 1703 of *LNCS*. Springer, 1999.

[VB00] Miroslav N. Velev and Randal E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *DAC*. ACM, 2000.