Formal Verification of a Processor with Memory Management Units

Dissertation

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

Iakov Dalinger

dalinger@wjpserver.cs.uni-sb.de

Saarbrücken, Juni 2006

Tag des Kolloquiums:	9. Juni 2006
Dekan:	Prof. Dr.-Ing. Thorsten Herfet
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Gerhard Weikum
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Peter-Michael Seidel
akademischer Mitarbeiter:	Dr. Mark Hillebrand

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im Juni 2006

Danksagung

Meiner Frau Yulia und meinen Eltern danke ich für ihre Unterstützung während der Jahre, in denen diese Arbeit entstanden ist.

Ich möchte mich bei Herrn Prof. Paul für die Betreuung und das interessante Thema dieser Arbeit bedanken.

Ich bedanke mich auch bei den Mitarbeitern des Lehrstuhls von Prof. Paul, insbesondere bei MARK HILLEBRAND und Sven Beyer, für vielen anregenden Diskussionen.

Abstract

In this thesis we present formal verification of a memory management unit which operates under specific conditions. We also present formal verification of a complex processor VAMP with support of address translation by means of a memory management unit. 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. The VAMP also supports precise internal and external interrupts. It is modeled on the gate level and verified with respect to its specification. Subject of this thesis is based on the formal proof of the VAMP without address translation [Bey05] and on paper and pencil specification, implementation, and correctness proof of a memory management unit [Hil05].

Kurzzusammenfassung

In dieser Dissertation stellen wir die formale Verifikation einer Memory Management Unit vor, welche nur unter bestimmten Operationsbedingungen korrekt arbeitet. Wir stellen auch die formale Verifikation des VAMP vor, eines komplexen Prozessors, der Adressübersetzung unterstützt. Der VAMP ist eine out-of-order 32-Bit RISC CPU mit DLX Instruktionssatz, vollständig IEEE-konformen Fließkommaeinheiten und einer Speichereinheit. Der VAMP unterstützt präzise interne und externe Interrupts. Er ist auf der Gatterebene modelliert und bezüglich einer formalen Spezifikation verifiziert. Diese Arbeit basiert auf dem formalen Beweis des VAMP ohne Adressübersetzung [Bey05] und auf der Papier-und-Bleistift Spezifikation, Implementierung, und dem Korrektheitsbeweis einer Memory Management Unit aus [Hil05].

Extended Abstract

In this thesis we report on the formal verification of an out-of-order processor with support of address translation. For this purpose a simple memory management unit was implemented and specified. The correctness proof for a memory management unit alone is simple, but depends on nontrivial operating conditions. The design of the memory management unit was inspired by dissertation of Mark Hillebrand [Hil05].

As the next step, we extend the microprocessor VAMP [BJK+03, BJK+05, Krö01, Bey05] without address translation with the memory management units, one for the address translation on instruction fetch and one for the address translation on load/store accesses. In order to have possibility to realize swap memory we also extended the VAMP with precise external interrupts. The verification is then split into two sub-steps, namely the correctness without interrupts and the correctness with interrupts.

Both implementation and verification of the processor and the memory management unit were carried out in the theorem proving system PVS [OSR92].

The VAMP is a pipelined out-of-order [Krö01] 32-bit RISC CPU with DLX instruction set, fully IEEE 754 [IEE85] compliant floating point units [Ber01, BJ01a, 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. An implementation of the Tomasulo algorithm with the floating point units, instruction fetch and a memory unit, as well as integrating precise interrupts and caches into the proof of the VAMP was done by Sven Beyer [Bey05].

The results of this work yields a formally verified gate-level implementation of the VAMP with support of address translation, with interrupts, and a cache memory interface with split instruction and data caches. Decomposition of the implementation to the variety of abstract levels allows to reason more effectively in opposite to arguing about the huge overall implementation.

We show overall correctness of the VAMP implementation with respect to its specification. We focus on the memory management unit, the data memory access, self-modifying code, instruction fetch and precise external interrupts. The results of this work are published in [DHP05]. Its importance is clear in the context of the verification of a complete computer system which consists of processor as hardware and operating system and several applications as the software counterpart. It is needed since common operat-

ing systems require paging and address translation on the hardware level in order to give each user program its own virtual memory.

Zusammenfassung

In dieser Arbeit berichten wir über die formale Verifikation eines out-of-order Prozessors mit Unterstützung für Adressübersetzung. Hierfür wurde eine einfache Memory Management Unit spezifiziert und implementiert. Der Korrektheitsbeweis dieser Einheit ist an sich einfach, hängt aber von nicht-trivialen Operationsbedingungen ab. Die Entwicklung der Memory Management Unit ist angelehnt an die Implementierung aus der Arbeit von Mark Hillebrand [Hil05].

In einem weiteren Schritt erweitern wir den Mikroprozessor VAMP ohne Adressübersetzung [BJK+03, BJK+05, Krö01, Bey05] mit Memory Management Units, eine für die Adressübersetzung bei Instruction Fetch und eine für die Adressübersetzung bei Lade-/ Speicherinstruktionen. Um die Implementierung von Auslagerungsspeicher zu ermöglichen, erweitern wir den VAMP für die Unterstützung von präzisen externen Interrupts.

Sowohl die Implementierung als auch die Verifikation des Prozessors und der Memory Management Unit erfolgen im Beweissystem PVS [OSR92].

Der VAMP ist eine gepipelinete out-of-order [Krö01] 32-Bit RISC CPU mit DLX Instruktionssatz, vollständig IEEE-754 [IEE85] konformen Fließkommaeinheiten [Ber01,BJ01a,Jac02a,Jac02b] für Operationen mit einfacher und doppelter Präzision, einer Speicher-Einheit mit einem Cache Memory Interface, delayed PC, und präzisen Interrupts. Die Entwicklung und die formale Verifikation des Tomasulo out-of-order Algorithmus [Tom67] und der Fließkommaeinheiten des VAMP basieren auf den Dissertationen von Daniel Kröning [Krö01] und Christian Jacobi [Jac02a]. Eine Implementierung des Tomasulo-Algorithmus mit Fließkommaeinheiten, Instruction Fetch, einer Speichereinheit mit Caches und internen präzisen Interrupts wurde von Sven Beyer entwickelt [Bey05].

Das Ergebnis beider Schritte ist eine auf Gatterebene formal verifizierte Implementierung des VAMP mit Unterstützung für Adressübersetzung, Interrupts und einem Cache Memory Interface mit getrennten Instruction und Data Caches. Die Zerlegung der Implementierung in abstrakte Ebenen erlaubt es, effizienter über die Korrektheit zu argumentieren als über die komplette Implementierung auf einmal.

Wir zeigen die vollständige Korrektheit der VAMP Implementierung bezüglich ihrer Spezifikation. Wir konzentrieren uns auf die Memory Management Unit, den Data Memory Access, selbst-modifizierenden Code, Instruction Fetch und präzise externe Interrupts. Die Ergebnisse dieser Arbeit sind in [DHP05] veroffentlicht. Die Wichtigkeit dieser Arbeit ergibt sich im Kontext der Verifikation eines vollständigen Computersystems, welches einen Prozessor als Hardware und ein Betriebssystem sowie verschiedene Applikationen als Software enthält. Standard-Betriebssysteme benötigen Paging und Adressübersetzung auf der Hardware-Ebene, um jeder Applikation ihren eigenen virtuellen Speicher zur Verfügung stellen zu können.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, there are many applications where different processors are used in life-critical devices, e.g., nuclear power stations, airplanes, trains, cars, or medical devices. Processor developers use diverse tests in order to find bugs in processors. Since non-exhaustive tests do not cover all possible cases of a processor behavior industrial processors may still contain bugs. 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. In addition, it scales better then simulation when one puts the correctness of several modules together.

The work presented here can be viewed as a link between two projects. The first is the *Verified Architecture Microprocessor* (VAMP) project [JK00, BBJ+02, BJK+03, BJK+05] where an instruction set architecture was specified, a complete microprocessor on the gate level was implemented, and formally verified in PVS [OSR92]. The second is the Verisoft project [Ver03], which is funded by the German federal government. One of the goals of the Verisoft project is to show the correctness of a system which has the VAMP as hardware and a compiler, an operating system, and several applications as software. The main distinction from previous processors is that the processor which we develop in this thesis has hardware support of address translation, which is very important in order to implement an multi-tasking operating systems.

The VAMP is a 32-bit RISC CPU with DLX instruction set [HP96b]. In addition to a memory unit with a cache memory interface [Bey05], the VAMP features a Tomasulo scheduler [Krö01], a fixed point unit, three floating point units [Ber01, BJ01a, Jac02a, Jac02b], and precise interrupts. For the formal verification of the VAMP, we focus on the memory unit and on instruction fetch. We prove the overall VAMP implementation correctness on the gate level with respect to a programmer's model that just executes one instruction in a step. For the formal verification the theorem-proving system PVS [OSR92] was used.

In this thesis we prove only the correctness of the hardware used in the Academic System of the Verisoft project. Note that software bugs may also be a reason of wrong system behavior. However, both hardware and software correctness proofs are necessary and the hardware verification is just a first step on the way to establish the correctness of the whole computer system.

Outline

In Chapter 1 we review the basic notation, which is used for the whole thesis. We also introduce the PVS system, virtual memory, and the general proof decomposition approach of this thesis.

In Chapter 2 we present a non-optimized construction of the memory management unit (*MMU*) which is a hardware supporting of virtual memory. We also prove its correctness under nontrivial operating conditions: the memory cells that are used for address translation must not change during a *MMU* request.

In Chapter 3 we establish the overall correctness of the VAMP with address translation. Note that in pipelined processors separate *MMU*s are used for instruction fetch and load / store. We present the specification and implementation of the VAMP. We show how the operating conditions for both *MMU*s can be guaranteed by a combination of hardware mechanisms and software conventions in order to exclude RAW hazards for fetch with address translation. Note that for the pipelined processor with address translation we have more RAW hazards. As software convention we require the usage of a `sync` instruction before a fetch from the modified memory location.

In the last Chapter 4 we summarize the results, discuss advantages and drawbacks of our system, present related work and possibilities for further work.

1.1 Notation

In this section we introduce some notation for the whole thesis. Since our work is based on Beyer's previous work [Bey05] we will use the same notation and abbreviations which were used in this work.

We define $\mathbb{N}$ as the set of natural numbers *including 0* and $\mathbb{N}^+ := \mathbb{N}\backslash\{0\}$. The set of integer numbers is denoted by $\mathbb{Z}$. We start with definitions for integer intervals.

Definition 1.1.1 *Let n, $m \in \mathbb{Z}$ be integer numbers. We define the following*

integer intervals:

$$\begin{aligned}
[n:m] &:= n,\ldots,m \\
]n:m] &:= n+1,\ldots,m \\
[n:m[&:= n,\ldots,m-1 \\
]n:m[&:= n+1,\ldots,m-1 \\
\mathbb{Z}_n &:= [0:n[\\
\mathbb{Z}_{\leq n} &:= [0:n] \\
\mathbb{Z}_{\geq n} &:= \mathbb{N} \setminus \mathbb{Z}_n.
\end{aligned}$$

Next we give definitions for bitvectors and arrays. We start with a standard definition of a word. We use the star ("*") to indicate that definitions and lemmas were copied from [Bey05].

Definition* 1.1.2 *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. The set of all finite words over Σ is given by $\Sigma^* := \bigcup_{n\in\mathbb{N}} \Sigma^n$. The **concatenation** of words is defined by*

$$\begin{aligned}
\circ : \Sigma^* \times \Sigma^* &\to \Sigma^* \\
a[n-1:0] \circ b[m-1:0] &= (a(n-1),\ldots,a(0),b(m-1),\ldots,b(0))
\end{aligned}$$

Instead of writing $\circ$ as infix operator, we also simply drop it, using $a[n-1:0]b[m-1:0]$ for concatenation.

Definition 1.1.3 *A **domain** is an alphabet. We abbreviate $\mathbb{B} := \{0,1\}$. A **bitvector** of length n is an array of length n over the domain $\mathbb{B}$.*

*Let b be **bitvector** of length n. We introduce the notation $b[k:l]$ for a **subbitvector** and $b[k]$ for k-th bit of the **bitvector** b, where $k \in \mathbb{Z}_n$ and $l \in \mathbb{Z}_k$.*

Note that the length for almost all bitvectors, which are used in this thesis, is divisible by 8. Note also that sometimes we will want to work only with one part (always 8 bits) of a bitvector. Therefore we introduce the next shorthand notation for such bitvectors in the following way:

Definition* 1.1.4 *For any $w \in \mathbb{B}^{8\cdot B}$ and $b < B$, we use the shorthand notation $|w|_b = w[8\cdot b + 7 : 8\cdot b]$ to select the b-th byte of bitvector w.*

Note that a bitvector of length 1 is a bit. For the whole thesis we identify the values 1 and 0 of a bit with the values *TRUE* and *FALSE* of a boolean correspondingly.

Definition* 1.1.5 *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 $\langle \cdot \rangle : \mathbb{B}^n \to \mathbb{Z}_{2^n}$ is bijective.*[1] *Thus, the function $\mathrm{bin}_n := \langle \cdot \rangle^{-1}$, $\mathrm{bin}_n : \mathbb{Z}_{2^n} \to \mathbb{B}^n$ that returns the **binary representation** of a natural number is well defined.*

Proposition* 1.1.6 *Let $a, b \in \mathbb{B}^n$, $m \in \mathbb{Z}_n$, and $k \in \mathbb{Z}_{2^m}$. The following statements hold:*

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

We will use these properties in order to prove our correctness criteria but we do not need to prove these properties because all of these proofs are included in the PVS `bitvectors` library.

Since our hardware implementation contains RAMs we now introduce a definition for RAM.

Definition* 1.1.7 *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 we can use λ-notation for arrays and RAMs to introduce unnamed functions. We only use λ-notation for functions definitions and we do not use real λ-calculus. For example the following equation trivially holds:

$$0 = \mathrm{bin}_n \langle \lambda_{i \in \mathbb{Z}_n} FALSE \rangle, \text{ where } n \in \mathbb{N}^+$$

Definition 1.1.8 *A function $inp : \mathbb{N} \to D_I$ is 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., $inp^t = inp(t)$.*

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

[1] This property is formally verified in the PVS standard library `bitvectors`, cf. Section 1.2.

that computes the configuration in the next cycle based on the current state and some input. We then denote the configuration 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.

Our proofs will not only depend on the signal behavior in the appointed cycle but we sometimes will use some additional information about the past or in the future.

Definition* 1.1.9 *Let S be a signal over the domain D, and P a predicate on D, and $t \in \mathbb{N}$ a cycle. We introduce the shorthand notation P^t for $P(S^t)$. We define a predicate indicating that P held in a cycle prior to t, namely $\exists_P^{last}(t) := \exists t' < t : P^{t'}$ and a another predicate indicating that P holds in the present cycle or in a future cycle, $\exists_P^{next}(t) := \exists t' \geq t : P^{t'}$.*

In case $\exists_P^{last}(t)$ holds, we define the last cycle where P held as $last_P(t) := \max\{t' < t : P^{t'}\}$. If $\exists_P^{next}(t)$ 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 $next(t)$.

Based on the last definition we can formulate some properties.

Proposition* 1.1.10 *Let S, D, P, and t be as in Definition 1.1.9. Then the following properties hold:*

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

1.2 The PVS System

There exist several systems which provide an integrated environment for the development and analysis of formal specifications. One of these systems is the *Prototype Verification System* [OSR92] (PVS). PVS consists of a specification language, a number of predefined libraries, a theorem prover and examples that illustrate different methods of using the system in several application areas. Since our work builds on previous work done in PVS we use the same verification environment.

The specification language of PVS is based on classical, typed higher-order logic. The PVS contains a `bitvectors` library. This is important for us since in hardware implementation we use signals which are represented in PVS as bitvectors. The `bitvectors` library provides a great variety of lemmas relating natural numbers and bitvectors. The conversion between natural numbers and bitvectors is possible with the `bv2nat` and `nat2bv` functions. In our description we define the set of PVS bitvectors `bvec[n]` as $\mathbb{B}^n$ and one bit as $\mathbb{B}$. Logical operations in the PVS model basics gates in the hardware circuits, i.e. *AND*, *OR*, *NOT*, etc.

In this interactive theorem prover all standard techniques which one can use in paper-and-pencil proof are present, i.e. natural induction, case splitting, skolemization, applications of lemmas, etc.

We do not present all of the PVS lemmas which were used in order to prove the processor correctness. The reason for this is that some lemmas are intuitively obvious but are necessary for the formal PVS theories. Moreover, the proofs of non-trivial lemmas we present in mathematical language omitting some formal details and hiding PVS syntax. Still, the structure of lemmas and proofs in this thesis straightforwardly reflects the structure of lemmas in PVS.

1.3 Basics Circuits

In this section we describe all of basics circuits which are used as macros later on in lemmas which we present in this thesis. All these circuits were proven correct before in [BJK01].

Definition 1.3.1 *Let $n \in \mathbb{N}^+$. We define the specification for some parametrized circuits:*

- *An* **n-bit adder** *is a circuit $Adder_n$ computing the function*

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

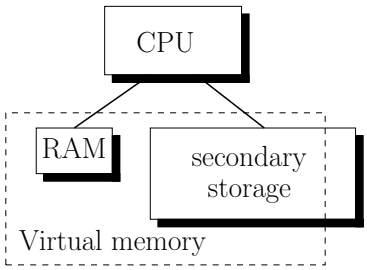

Figure 1.1: Organization of Virtual Memory

- *An* **n-*bit adder and subtractor*** *is a circuit* Add_sub_n *computing the function*

$$add\_sub_n : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^n,$$
$$add\_sub_n(a, b, sub) = (neg, sum), \ \textit{where}$$

$$\begin{aligned} add\_sub_n(a,b,sub).neg &:= (\langle a \rangle - \langle b \rangle < 0), \\ add\_sub_n(a,b,sub).sum &:= \begin{cases} \mathrm{bin}_{n+1}((\langle a \rangle - \langle b \rangle) \bmod n) & \textit{if } sub \\ \mathrm{bin}_{n+1}(\langle a \rangle + \langle b \rangle) & \textit{otherwise.} \end{cases} \end{aligned}$$

Note that we do not describe implementation and correctness of these circuits in the thesis. Implementations and correctness proofs for these circuits are available in the PVS `basic` library presented in [BJK01].

1.4 Virtual Memory

Nowadays, software requires more physical memory than the hardware provides. The most common solution to this problem is the use of virtual memory as shown in Figure 1.1.

The virtual memory is a memory which is created using the secondary storage in order to simulate additional random-access memory. The addressable space of the secondary storage is available to the user of a computer system in which virtual addresses are mapped into physical addresses.

The virtual address space is divided into pages, usually a few kilobytes large. Each virtual address is split into a virtual page index and a byte index which is an offset within the page. For a page size of 2^n bytes, the byte index consist of the n least significant bits of a virtual address.

Virtual memory is usually much larger than physical memory, making it possible to run programs for which the total code plus data size is greater than the amount of RAM available. The excess is stored on secondary storage, usually on a hard disk. A page is copied from disk to RAM, "paged in", when an attempt to access it is made and it is not in the RAM. This

is known as "demand paging". It is performed by a collaboration between the CPU, the memory management unit, which we describe later, and the operating system kernel. The program is unaware of this collaboration, it just sees a large address space, only a part of which corresponds to physical memory. In this case programs may, of course, run slower.

A memory management unit is a piece of hardware which performs address translation. It is located between the processor and the physical memory which usually contains a cache and main memory. The *MMU* maps the virtual page index to a physical page index and the byte index is left unchanged. The memory contains a page table which is indexed by the page index (more details in Chapter 2, Section 2.1.2). Each page table entry (PTE) contains the physical page index corresponding to the virtual one. A physical page index is combined with the page offset (byte index) to give the complete physical address.

For each page the PTE may also contain the following information:

- whether the page has been written to,
- when it was last used,
- whether a process may read and write it,
- whether the page is stored in the physical memory or on the hard disk.

In case no physical memory has been allocated to a given virtual page the *MMU* signals a "page fault" to the CPU. The operating system will then try to find a unused page in RAM and load this page into RAM. If there is no free RAM it chooses an occupied page, using some replacement algorithm, and saves it to a disk.

In this thesis we prove the correctness of a system which contains three components: a processor, a memory management unit, and a physical memory. Note that our model does not contain any secondary storage, but the processor has external interrupts which can help to extend the model with the secondary storage [HIP05].

1.5 Proof Decomposition

We now give an overview on the overall correctness proof of the VAMP processor with support of the address translation. Since our work is in extension of the previous work of our chair we base this thesis on the previous proof decomposition which was described in [Bey05]. It has the modules which are presented in Figure 1.2.

Note that we want to use as many lemmas as possible from the previous proof without changes. Therefore we proceed as follows:

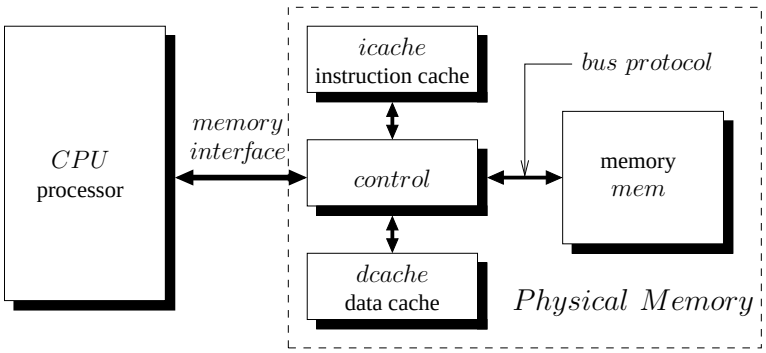

Figure 1.2: Proof Decomposition of the VAMP without Address Translation

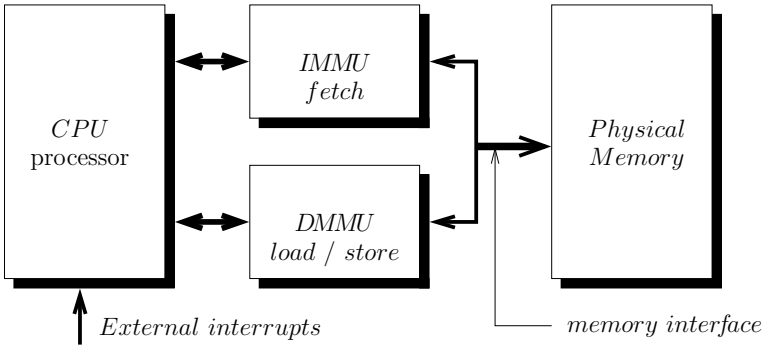

Figure 1.3: Proof Decomposition of the VAMP with Address Translation

- First, in order to extend the VAMP with two new modules we need to change the CPU module because we need some additional registers. We also need to extend the CPU module with the external interrupts with the help of which we can work with external devices, e.g. a hard disk as secondary storage.

- Second, the memory interface which was developed in the previous work differs in no way from standard double-port RAM interface (one port for instruction and one for data) and, of course, we want to keep this interface because in this case we can use the CPU module together with any RAM which has the same interface. Therefore we try not to change the memory interface at all. In this case, of course, we can use all the old proofs for the module *Physical Memory*.

A model of a processor with address translation is presented in Figure 1.2. There are two additional modules in this model:

- the Instruction *MMU* which performs translated and untranslated memory accesses for instruction fetch.

Signal	Description
Memory interface input to *DMMU*	
$dadr[a-1:0]$	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
$mbw[B-1:0]$	selects bytes of the data word to be written
Memory interface input to *IMMU*	
$iadr[a-1:0]$	word address of the instruction access
imr	signals instruction read access
$clear$	initializes memory interface (the signal comes directly from CPU)
Memory interface output from *IMMU*	
$ibusy$	signals pending instruction access
$inst[8 \cdot B-1:0]$	read data on finished instruction access
Memory interface output from *DMMU*	
$dbusy$	signals pending data access
$dout[8 \cdot B-1:0]$	read data on finished data access

Table 1.1: The Memory Interface (between *MMU*s and Physical Memory)

- the Data *MMU* which performs translated and untranslated memory accesses for load / store instructions.

1.5.1 The Memory Interface Layer

We keep a definition of the memory interface which was described in [Bey05]. We have only one difference: now the initiator of memory request is the *DMMU* or the *IMMU* but not the CPU. We can also divide the memory interface into two parts. The first part will be an interface with the *IMMU* and the second with the *DMMU*. Note that signal *clear* still comes directly from the processor (not from one of the *MMU*s).

Definition* 1.5.1 *A **memory system** M_I for a pipelined microprocessor is a pice of hardware with inputs and outputs according to Table 1.1, where 2^a is the number of cells stored in the memory for $a \in \mathbb{N}$ and B is the number of bytes stored per memory cell. Note that Beyer calls M_I as memory interface.*

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

- $\forall t \in \mathbb{N} : clear^t \iff t = 0$

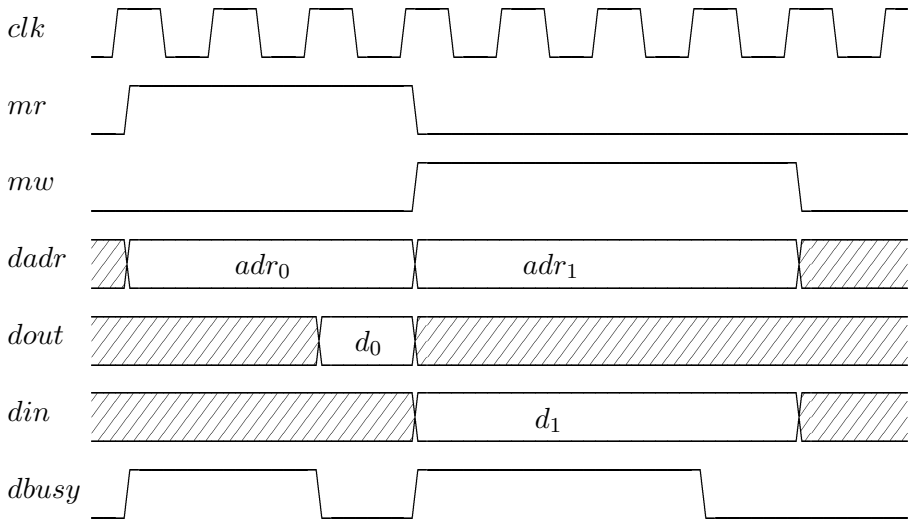

Figure 1.4: Timing of the Memory Interface for *DMMU*

- $\forall t \in \mathbb{N}^+ : \neg mw^t \vee \neg mr^t$
- $\forall t \in \mathbb{N}^+ : (mr^t \vee mw^t) \wedge dbusy^t \implies in_d^{t+1} = in_d^t,$
  $where\ in_d \in \{dadr, din, mw, mr, mbw\}$
- $\forall t \in \mathbb{N}^+ : imr^t \wedge ibusy^t \implies in_i^{t+1} = in_i^t,\ where\ in_i \in \{iadr, imr\}$

A partial timing diagram of the memory interface is depicted in Figure 1.4. The *DMMU* starts a request in a cycle t (one of two signals mr^t for a read or mw^t for a write is active). The address $dadr^t$, data din^t, and mbw^t keep their value during the whole duration of a request until cycle $t' \geq t$ for which $dbusy^{t'}$ is inactive.

The timing diagram between *IMMU* and the memory is similar to the previous one. Note that we have only read accesses to the *IMMU*.

We now define the correctness of the memory interface with valid inputs from *DMMU* and *IMMU*. The memory interface is correct if it is live and consistent according to the following definition.

Definition* 1.5.2 *Let* $init\_mem \in (\mathbb{B}^{8 \cdot B})^{2^a}$ *be the initial memory content of a memory interface.*

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

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

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

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

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

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

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

1.5.2 The CPU Interface Layer

We now define the last interface according to Figure 1.3. This interface is divided into two parts, the first part is the interface between the CPU and the *DMMU* and the second is the interface between the CPU and the *IMMU*.

Definition 1.5.3 *A* ***CPU interface*** CPU_I *for a pipelined microprocessor is a circuit with inputs and outputs according to Table 1.2.*

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

- $\forall t \in \mathbb{N}^+ : \neg mw^t \vee \neg mr^t$
- $\forall t \in \mathbb{N}^+ : (mr^t \vee mw^t) \wedge dbusy^t \implies in_d^{t+1} = in_d^t$
  $where\ in_d \in \{dadr, dout, mw, mr, mbw, dpto, dptl\}$
- $\forall t \in \mathbb{N}^+ : imr^t \wedge ibusy^t \implies in_i^{t+1} = in_i^t,$
  $where\ in_i \in \{iadr, imr, ipto, iptl\}$

A partial timing diagram of the CPU_I interface is depicted in Figure 1.5. The timing diagram between *IMMU* and CPU is the similar to the previous one, in the case of read access.

In the following chapter we present the design and formal verification of the *MMU* which is used in Chapter 3 as *instruction MMU* and *data MMU*. In Chapter 3, we specify a processor model, then we build hardware for it and verify the data consistency of the whole VAMP processor.

Signal	Description
Interface output from CPU	
$dadr[a-1:0]$	word address of the data access
$dout[8 \cdot B-1:0]$	data word to be written
mw	signals data write access
mr	signals data read access
$dpto[19:0]$	data page table origin
$dptl[19:0]$	data page table length
$dmode$	$system/user$ mode for the data access
$mbw[B-1:0]$	selects bytes of the data word to be written
$iadr[a-1:0]$	word address of the instruction access
imr	signals instruction read access
$ipto[19:0]$	instruction page table origin
$iptl[19:0]$	instruction page table length
$imode$	$system/user$ mode for the instruction access
$clear$	initializes memory interface
Interface input to CPU	
$ibusy$	signals pending instruction access
$inst[8 \cdot B-1:0]$	read data on finished instruction access
$iexcp$	signals exception on finished instruction access
$dbusy$	signals pending data access
$din[8 \cdot B-1:0]$	read data on finished data access
$dexcp$	signals exception on finished data access

Table 1.2: The CPU_I Interface (between CPU and MMUs)

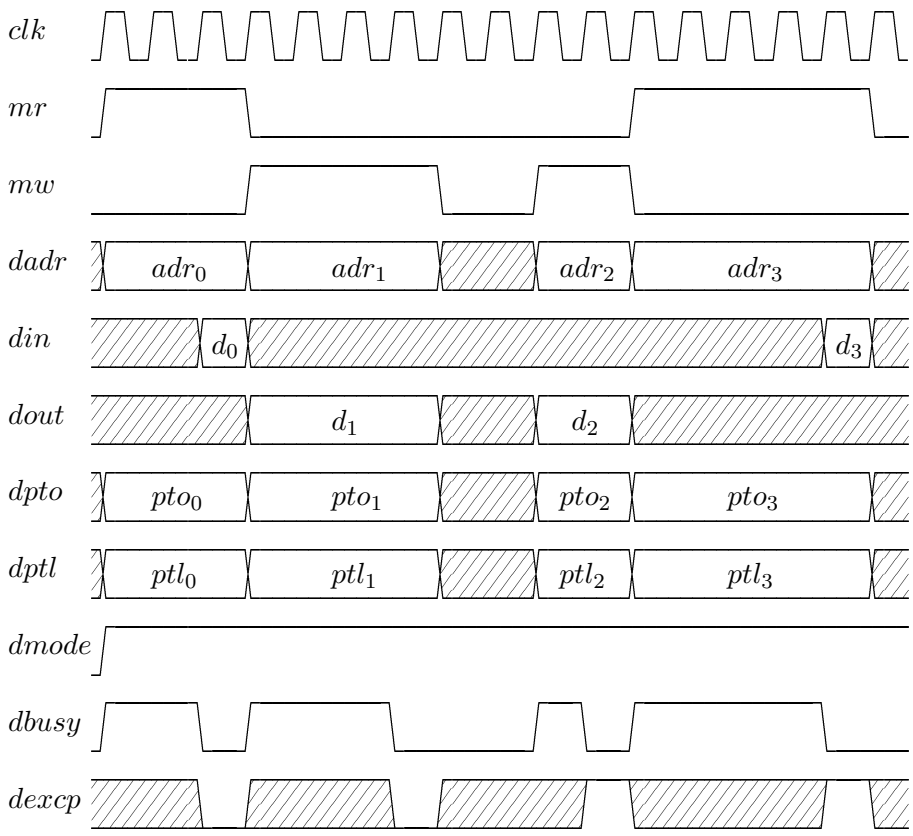

Figure 1.5: Timing of the Interface between CPU and $DMMU$

Chapter 2

The Memory Management Unit

The work presented in this chapter is based on Hillebrand's work [Hil05]. In this chapter we introduce the specification and the implementation of a memory management unit and we formally verify the *MMU*. As we already know, the *MMU* is a hardware supporting the implementation of the virtual memory and it is located between the processor and the memory.

2.1 Specification of the MMU

Since the *MMU* communicates with processor and memory we specify both interfaces. The set of all signals in an interface is called interface observation. We also specify the memory configuration.

We define the configuration C_{spec} for the specification of the *MMU*. This configuration has the following components:

- $Iobs_p$ – the processor interface observations.
- $Iobs_m$ – the memory interface observations.
- $Memory$ – the configuration of the physical memory.

An element $c \in C_{spec}$ of the *MMU* specification configuration is triple:

$$c = (iobs_p \in Iobs_p, iobs_m \in iobs_m, mem \in Memory)$$

The processor interface observation $iobs_p$ is a 12-tuple:

$$iobs_p = (t, mw, mr, mbw, addr, dout, din, pto, ptl, excp, busy, reset)$$

We start to describe the components which are inputs to the *MMU* as follows:

- $t \in \mathbb{B}$ – the memory access type, if this flag is set we have a translated memory access, otherwise an untranslated memory access.
- $mw \in \mathbb{B}$ – if this flag is set we have a write access.
- $mr \in \mathbb{B}$ – if this flag is set we have a read access.
- $mbw \in \mathbb{B}^8$ – the memory byte write signals. As every memory request accesses a double word, the signal mbw indicates which bytes are written. We use this component only in the case of a write access.
- $addr \in \mathbb{B}^{29}$ – data address of the memory access. Depending on the flag t, the address is either a virtual or a physical one.
- $dout \in \mathbb{B}^{64}$ – the data to be stored in the memory. This data will be read for write accesses only.[1]
- $pto \in \mathbb{B}^{20}$ – the page table origin is used only for translated memory operations. The address $\langle pto \circ 0^{12} \rangle$ is the first address of the page table.
- $ptl \in \mathbb{B}^{20}$ – the page table length is used only for translated memory accesses and shows the size of the page table.
- $reset \in \mathbb{B}$ – reset flag is active in case of processor reset.

The rest of the components are outputs from the *MMU*:

- $din \in \mathbb{B}^{64}$ – the data read from the memory. The data will be used for read accesses only.
- $excp \in \mathbb{B}$ – exception flag, this flag is set in case the memory operation is terminated abnormally.
- $busy \in \mathbb{B}$ – the busy flag signals that the *MMU* is busy to the processor.

The memory interface observation $iobs_m$ is a 7-tuple:

$$iobs_m = (mw, mr, mbw, addr, dout, din, busy)$$

We now describe the components which are inputs to the memory:

- $mw \in \mathbb{B}$ – if this flag is set we have a write access.
- $mr \in \mathbb{B}$ – if this flag is set we have a read access.
- $mbw \in \mathbb{B}^8$ – the memory byte write signals. As every memory request accesses double word, the signal mbw indicates which bytes are written. We use this component only in the case of a write access.

[1] In this chapter, we use the nomenclature *in* and *out* from a processor's point of view, e.g. *dout* is output from the processor's point of view.

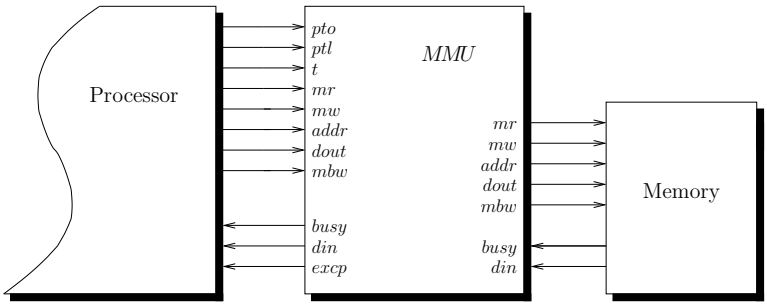

Figure 2.1: Interfaces of the *MMU*

- $addr \in \mathbb{B}^{29}$ – data address in the physical memory.
- $dout \in \mathbb{B}^{64}$ – the data to be written into the memory.

The rest of the components are outputs from the memory:

- $din \in \mathbb{B}^{64}$ – the data read from the memory.
- $busy \in \mathbb{B}$ – the busy flag signals that the memory is busy to the *MMU*.

Both interfaces are shown in Figure 2.1.

Definition 2.1.1 *Depending on the read and write signals we define a new signal for interface* $iobs_p$ *by*

$$iobs_p.req = iobs_p.mw \vee iobs_p.mr$$

and similarly for interface $iobs_m$ *by*

$$iobs_m.req = iobs_m.mw \vee iobs_m.mr.$$

Note also that we use short notation $iobs_p.inputs$ *(*$iobs_m.inputs$*) for processor (memory) inputs, i.e. all signals which are inputs for the MMU (physical memory). We also use similar abbreviations* $iobs_p.outputs$ *(*$iobs_m.outputs$*) for processor (memory) outputs, i.e. all signals which are outputs for the MMU (physical memory).*

Note that we use record notation for both interfaces, e.g. component *din* of $iobs_m$ is denoted by $iobs_m.din$ and $iobs_p.pto$ denote component *pto* of $iobs_m$.

Definition 2.1.2 *We call a memory configuration mem a function that maps* 29*-bit addresses to* 64*-bit data, i.e. the memory is organized in* 2^{29} *double words:*

$$mem \in Memory = \{f : \mathbb{Z}_{2^{29}} \to \mathbb{B}^{64}\}$$

A function that maps times $t \in \mathbb{N}$ to the configuration of the *MMU* specification $f(t) \in C_{spec}$ is called a trace.

Definition 2.1.3 *We define the set of all specification traces as follows:*

$$Trace = \{f : \mathbb{N} \to C_{spec}\}$$

2.1.1 Assumption for the MMU

We make assumptions on the signals which are inputs of the *MMU*. Later, in Chapter 3 we need to prove these assumptions when we integrate the *MMU* into the processor according to Definition 1.5.1 and Definition 1.5.3.

Definition 2.1.4 *Let s be any signal from interfaces $iobs_p$ or $iobs_m$. We use the short notation s_x for $iobs_x.s$ where x denotes one of two interfaces p for processor and m for memory.*

We now can define the following properties for the processor interface. We formulate this properties as predicates on traces. For the whole section $trc \in Trace$ denotes a trace.

Definition 2.1.5 *We call the signals read and write of the processor interface mutually exclusive if the following predicate holds:*

$$\begin{aligned} &p\_mr\_mw\_mutexc(trc) := \\ &\qquad \forall t \in \mathbb{N} : \neg(trc(t).mr_p \wedge trc(t).mw_p) \end{aligned}$$

Definition 2.1.6 *We call the input signals of the processor interface stable if the following predicate holds:*

$$\begin{aligned} &p\_req\_is\_stable(trc) := \\ &\forall t \in \mathbb{N} : trc(t).req_p \wedge trc(t).busy_p \implies \\ &\qquad trc(t+1).inputs_p = trc(t).inputs_p \end{aligned}$$

Now we define a predicate for the processor request.

Definition 2.1.7 *We define the predicate is_req_proc for a processor request in the following way:*

$$\begin{aligned} &is\_req\_proc(t, t', trc) := \\ &\qquad (t > 0 \implies \neg trc(t-1).busy_p) \wedge \\ &\qquad (t \leq t') \wedge trc(t).req_p \wedge \neg trc(t').busy_p \wedge \\ &\qquad \forall t'' \in [t : t'[\ : \ trc(t'').busy_p \end{aligned}$$

The signal $busy_p$ is active during the whole processor request but the last cycle. Since we can only change state of $inputs_p$ when $busy_p$ is inactive (Definition 2.1.6) we can prove in the following lemma that during a processor request all the processor inputs are stable.

Lemma 2.1.8 *All of the inputs in the processor are stable processor request:*

$$\begin{aligned}\forall t, t', t'' \in \mathbb{N} : t \leq t' \leq t'' \implies \\ (is\_req\_proc(t, t'', trc) \wedge p\_req\_is\_stable(trc)) \implies \\ trc(t).inputs_p = trc(t').inputs_p\end{aligned}$$

Proof: Let $t \leq t' \leq t''$. We show the claim by induction on t'. For the base case $t = t'$ there is nothing to show. For the induction step $t' \rightarrow t' + 1$ we have that

$$\begin{aligned}&t' < t'' \wedge \\ &is\_req\_proc(t, t'', trc) \wedge \\ &p\_req\_is\_stable(trc) \wedge \\ &trc(t).inputs_p = trc(t').inputs_p\end{aligned}$$

We have to show that

$$trc(t).inputs_p = trc(t' + 1).inputs_p.$$

With the help of the induction hypothesis we can conclude that:

$$trc(t).iobs_p.inputs = trc(t').inputs_p$$

Since we know that $(trc(t').mr_p \vee trc(t').mw_p) \wedge trc(t').busy_p$ holds we conclude with the help of $p\_req\_is\_stable(trc)$ at the time t':

$$trc(t').inputs_p = trc(t' + 1).inputs_p$$

This finishes the proof. □

Finally we define the predicate of the whole processor interface correctness criteria as follows:

Definition 2.1.9 *Processor inputs are called correct if read and write signals are mutually exclusive and they are stable. Formally:*

$$\begin{aligned}good\_p\_interface(trc) := \\ p\_mr\_mw\_mutexc(trc) \wedge \\ p\_req\_is\_stable(trc)\end{aligned}$$

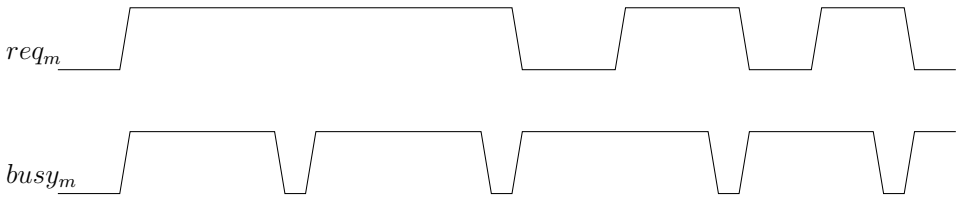

Figure 2.2: Example of the Behavior of the Signal $busy_m$

For the interface between the *MMU* and the memory we make some assumptions which are based on properties of signals of Definition 1.5.1. We now define a predicate for a memory request similar to the predicate is_req_proc.

Definition 2.1.10 *We define the predicate is_req_mem in the following way:*

$$\begin{aligned} is\_req\_mem(t, t', trc) := \\ & (t > 0 \implies (\neg trc(t-1).busy_m \vee \neg trc(t-1).req_m)) \wedge \\ & t \leq t' \wedge trc(t).req_m \wedge \neg trc(t').busy_m \wedge \\ & \forall t'' \in [t : t'[\ : \ trc(t'').busy_m \end{aligned}$$

Note that this predicate is defined slightly different then is_req_proc. In case if $t > 0$ a memory request can also start in cycle t when $busy_m^{t-1} \wedge \neg req_m^{t-1}$. Therefore for start of request in cycle $t-1$ we do not depend on signal $busy_m$ in the case when we do not have any request to the memory (see Figure 2.2), i.e. for this definition we can also use an memory interface in which signal *busy* is undefined when we do not have any requests.

Definition 2.1.11 *We call memory inputs live if the predicate $m\_liveness(trc)$ holds, i.e. formally we have*

$$\begin{aligned} m\_liveness(trc) := \\ & \forall t \in \mathbb{N} : trc(t).req_m \implies \\ & \exists t' \in \mathbb{Z}_{\geq t} : \neg trc(t').busy_m \end{aligned}$$

We split the consistency of the memory into three assumptions. The first assumption covers the behavior of the memory except last cycle of a memory request and the other two cover terminating read and write accesses, respectively. Note that we assume here that a *MMU* has exclusive access to the memory.

Definition 2.1.12 *The predicate $m\_ack\_write(trc)$ holds if the memory does not change if there is no read or write signal input to the memory or*

the memory is busy:

$$\begin{aligned} &m\_ack\_write(trc) := \\ &\quad \forall t \in \mathbb{N} : (\neg trc(t).req_m \vee trc(t).busy_m) \\ &\quad\quad \implies trc(t).mem = trc(t+1).mem \end{aligned}$$

Definition 2.1.13 *The predicate $m\_read\_consist(trc)$ holds if at the end of any memory read access we have the correct data on the output from the memory and during the last cycle of access the memory does not change:*

$$\begin{aligned} &m\_read\_consist(trc) := \\ &\quad \forall t, t' \in \mathbb{N} : is\_req\_mem(t, t', trc) \wedge trc(t).mr_m \implies \\ &\quad trc(t').din_m = trc(t').mem[\langle trc(t).addr_m \rangle] \wedge \\ &\quad trc(t').mem = trc(t'+1).mem \end{aligned}$$

Definition 2.1.14 *The predicate $m\_write\_consist(trc)$ holds if at the end of any memory write access the memory is updated in the following way:*

$$\begin{aligned} &m\_write\_consist(trc) := \\ &\quad \forall t, t' \in \mathbb{N} : is\_req\_mem(t, t', trc) \wedge trc(t).mw_m \implies \\ &\quad \forall a \in \mathbb{B}^{29} : (a \neq trc(t).addr_m \implies \\ &\quad\quad trc(t').mem[\langle a \rangle] = trc(t'+1).mem[\langle a \rangle]) \wedge \\ &\quad \forall b \in \mathbb{Z}_8 \left| trc(t'+1).mem[\langle trc(t).addr_m \rangle] \right|_b = \\ &\quad\quad \begin{cases} |trc(t).dout_m|_b & \text{if } trc(t).mbw_m[b] \\ |trc(t').mem[\langle trc(t).addr_m \rangle]|_b & \text{otherwise} \end{cases} \end{aligned}$$

Finally, we define the predicate for overall memory interface correctness in the following way:

Definition 2.1.15 *We call a memory inputs and configuration correct if the memory is consistent and live:*

$$\begin{aligned} &good\_m\_interface(trc) := \\ &\quad\quad m\_liveness(trc) \wedge \\ &\quad\quad m\_ack\_write(trc) \wedge \\ &\quad\quad m\_read\_consist(trc) \wedge \\ &\quad\quad m\_write\_consist(trc) \end{aligned}$$

2.1.2 Guarantees of the MMU

In this section we will define properties which model correct *MMU* behavior. We start with liveness.

Definition 2.1.16 *The predicate $p\_liveness(trc)$ holds if all of the processor requests end in finite time, i.e., after setting read or write signals on the inputs in the MMU the MMU eventually releases busy:*

$$p\_liveness(trc) := \\ \forall t \in \mathbb{N} : trc(t).req_p \implies \exists t' \in \mathbb{Z}_{\geq t} : \neg trc(t').busy_p$$

We also have to guarantee according to the Definition 1.5.1 that read and write signals are not set simultaneously.

Definition 2.1.17 *The predicate $m\_mr\_mw\_mutexc(trc)$ holds when the signals read and write to the memory be mutually exclusive:*

$$m\_mr\_mw\_mutexc(trc) := \\ \forall t \in \mathbb{N} : \neg(trc(t).mr_m \wedge trc(t).mw_m)$$

The *MMU* keeps the inputs for the memory stable during requests.

Definition 2.1.18 *The predicate $m\_req\_is\_stable(trc)$ holds if all input signals of the memory interface are called stable if the memory is busy in the same cycle:*

$$m\_req\_is\_stable(trc) :=$$

$$\forall t \in \mathbb{N} : (trc(t).mr_m \vee trc(t).mw_m) \wedge trc(t).busy_m) \implies$$

$$\begin{array}{lcl} trc(t).mr_m & = & trc(t+1).mr_m \wedge \\ trc(t).mw_m & = & trc(t+1).mw_m \wedge \\ trc(t).addr_m & = & trc(t+1).addr_m \wedge \\ trc(t).mbw_m & = & trc(t+1).mbw_m \wedge \\ trc(t).din_m & = & trc(t+1).din_m \end{array}$$

Based on the definition of a memory request and the previous property we can prove that inputs remains stable in analogy to Lemma 2.1.8.

Lemma 2.1.19 *All of the inputs in the memory are stable as long as the memory is busy:*

$$\forall t, t', t'' \in \mathbb{N} : t \leq t' \leq t'' \implies \\ (is\_req\_mem(t, t'', trc) \wedge m\_req\_is\_stable(trc)) \implies \\ trc(t).inputs_m = trc(t').inputs_m$$

We omit the proof of this lemma because it is similar to the proof of Lemma 2.1.8.

Now we specify all the *MMU* operations. The *MMU* should realize four types of memory operations as requested by the processor:

- Untranslated read and write – directly access the memory using 29-bit physical addresses.
- Translated read and write – access the memory using 29-bit *virtual* addresses. However, the translation of a virtual address can fail, in which case the memory operation cannot be executed. In this case the *MMU* should signal an exception to the processor.

We start with the definition for the untranslated read. In this case we do not translate the address from the processor. The data of the memory from the processor address is returned. We do not have any exception.

Definition 2.1.20 *An untranslated read is specified by:*

$$
\begin{aligned}
&untr\_read(trc) := \\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge \\
&\quad trc(t).mr_p \wedge \neg trc(t).iobs_p.t \implies \\
&\qquad trc(t').din_p = trc(t').mem[\langle trc(t).addr_p \rangle] \wedge \\
&\qquad trc(t').mem = trc(t'+1).mem \wedge \neg trc(t').iobs_p.excp
\end{aligned}
$$

In the case of an untranslated write access we only write into the memory at the processor address the data from the processor. Depending on the memory byte write signal it could be one, two, four, or eight bytes. We also do not have any exception.

Definition 2.1.21 *An untranslated write access is specified by:*

$$
\begin{aligned}
&untr\_write(trc) := \\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge \\
&\quad trc(t).mw_p \wedge \neg trc(t).iobs_p.t \implies \\
&\quad \forall a \in \mathbb{B}^{29} : (a \neq trc(t).addr_p \implies \\
&\qquad\qquad trc(t').mem[\langle a \rangle] = trc(t'+1).mem[\langle a \rangle]) \wedge \\
&\quad \neg trc(t').iobs_p.excp \wedge \\
&\quad \forall b \in \mathbb{Z}_8 \left| trc(t'+1).mem[\langle trc(t).addr_p \rangle] \right|_b = \\
&\qquad \begin{cases} |trc(t).dout_p|_b & \text{if } trc(t).mbw_p[b] \\ |trc(t').mem[\langle trc(t).addr_p \rangle]|_b & \text{otherwise} \end{cases}
\end{aligned}
$$

For the translated operations we have to define an additional function for address translation. This function is called the *decodeitr* — *decode implementation translation* function:

$$decodeitr : \mathbb{B}^{20} \times \mathbb{B}^{20} \times Memory \times \mathbb{B} \times \mathbb{B}^{32} \to \mathbb{B} \times \mathbb{B}^{32}$$

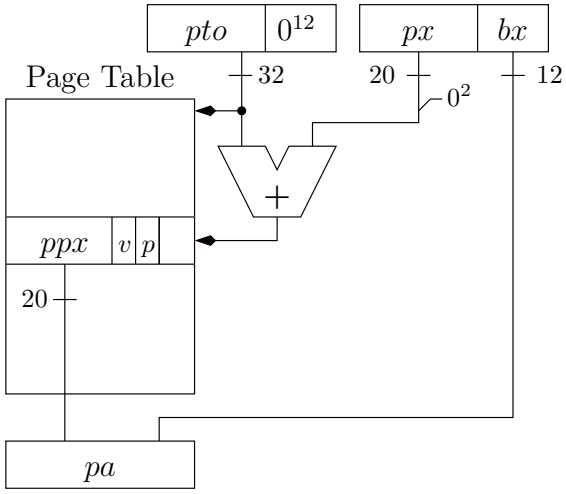

Figure 2.3: Address Translation for the *virtual address*

Figure 2.3 shows the principle of address translation, which is done by *decodeitr* function. This function takes the following five inputs

- $pto \in \mathbb{B}^{20}$ – the page table origin,
- $ptl \in \mathbb{B}^{20}$ – the page table length,
- $mem \in Memory$ – the memory configuration,
- $mw \in \mathbb{B}$ – the type of request read or write,
- $va \in \mathbb{B}^{32}$ – is the virtual address.

The function *decodeitr* has two outputs $excp \in \mathbb{B}$ and $pa \in \mathbb{B}^{32}$ where *excp* indicates a translation exception and *pa* is the *physical address*, i.e., translated memory address. For the calculation of *excp* and *pa* we define some intermediate values. The virtual address is decomposed into a page index $px \in \mathbb{B}^{20}$ and a byte index $bx \in \mathbb{B}^{12}$:

$$va = px \circ bx$$

The page table consists of page table entries *pte*. Each *pte* is 4 bytes wide. The number of page table entries in the page table is equal to $ptl + 1$. Based on *pto* and *px* we define the page table entry address $ptea \in \mathbb{B}^{32}$ as follows:

$$ptea = (\langle pto \circ 0^{12} \rangle + \langle px \circ 0^2 \rangle) \bmod 2^{32}$$

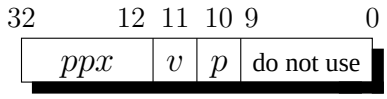

Figure 2.4: Page Table Entry

Based on *ptea* we define the page table entry $pte \in \mathbb{B}^{32}$ as follows

$$pte = \begin{cases} memory(\lfloor ptea/8 \rfloor)[31:0] & \text{if } ptea \bmod 8 = 0 \\ memory(\lfloor ptea/8 \rfloor)[63:32] & \text{otherwise} \end{cases} \tag{2.1}$$

where $a \in \mathbb{R}$ is $\lfloor a \rfloor$ is rounding down to the next integer.

The page table entry $pte \in \mathbb{B}^{32}$ (see Figure 2.4) consists of several fields:

- $ppx = pte[31:12]$ – the physical page index,
- $v = pte[11]$ – the valid bit which indicates whether the page is in the physical memory or on the secondary storage,
- $p = pte[10]$ – the write protection bit,
- $pte[9:0]$ – we do not use the last ten bits.

We have an exception in the following cases:

- the page index is larger then the page-table length, i.e., the access would be outside the page table,
- we have a write access and the page is protected,
- the page is not in the physical memory, as indicated by the valid bit.

Therefore *excp* is given by the following equation:

$$\begin{aligned} lexcp &= \langle px \rangle > \langle ptl \rangle \\ pteexcp &= (mw \wedge p \vee \neg v) \\ excp &= lexcp \vee pteexcp \end{aligned}$$

We calculate the *physical address* so that in the case of $\neg excp$, we compute *pa* as concatenation of the physical page index and byte index. Otherwise, *pa* is set to 0^{32}:

$$pa = \begin{cases} 0^{32} & \text{if } excp \\ ppx \circ bx & \text{otherwise} \end{cases}$$

We use the previously defined variables to define the result of a translation r_t of a processor request starting in cycle t as follows:

$$
\begin{aligned}
&\mathit{Let}\ \ d = \mathit{decodeitr}(trc(t).pto_p,\\
&\qquad\qquad\qquad\qquad trc(t).ptl_p,\\
&\qquad\qquad\qquad\qquad trc(t).iobs_p.mem,\\
&\qquad\qquad\qquad\qquad trc(t).mw_p,\\
&\qquad\qquad\qquad\qquad trc(t).addr_p \circ 0^3)\ \ \mathit{in}\\
&r\_t(t).pa = d.pa,\\
&r\_t(t).excp = d.excp,\\
&r\_t(t).ptea = ptea,\\
&r\_t(t).lexcp = lexcp,\\
&r\_t(t).pteexcp = pteexcp,\\
&r\_t(t).pte = pte.
\end{aligned}
$$

Now we can define both translated operations. We start with the translated read. If there is no exception, the data of the memory from the physical address is returned. Otherwise, exceptions are signaled to the processor.

Definition 2.1.22 *A translated read access is specified by:*

$$
\begin{aligned}
&tr\_read(trc) :=\\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge\\
&\quad trc(t).mr_p \wedge trc(t).iobs_p.t \implies\\
&\qquad trc(t').iobs_p.excp = r\_t(t).excp \wedge\\
&\qquad trc(t').mem = trc(t'+1).mem \wedge\\
&\qquad \neg r\_t(t).excp \implies trc(t').din_p = trc(t').mem[\langle r\_t(t).pa[31:3]\rangle]
\end{aligned}
$$

The last operation is translated write. In case of an exception the memory is not modified. If we do not have any exception we only write into the memory at the physical address the data from the processor.

Definition 2.1.23 *A translated write access is specified by:*

$$
\begin{aligned}
&tr\_write(trc) := \\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge \\
&\quad trc(t).mw_p \wedge trc(t).iobs_p.t \implies \\
&\qquad (r\_t(t).excp = trc(t').iobs_p.excp \wedge \\
&\qquad\quad (r\_t(t).excp \implies trc(t').mem = trc(t'+1).mem) \wedge \\
&\qquad\quad (\neg r\_t(t).excp \implies trc(t').addr_m = r\_t(t).pa[31:3] \wedge \\
&\qquad\qquad (\forall a \in \mathbb{B}^{29} : (a \neq r\_t(t).pa[31:3] \implies \\
&\qquad\qquad\qquad trc(t').mem[\langle a \rangle] = trc(t'+1).mem[\langle a \rangle])) \wedge \\
&\qquad\qquad \forall b \in \mathbb{Z}_8 : \left|trc(t'+1).mem[\langle r\_t(t).pa[31:3] \rangle]\right|_b = \\
&\qquad\qquad\quad \begin{cases} |trc(t).dout_p|_b & \text{if } trc(t).mbw_p[b] \\ |trc(t').mem[\langle r\_t(t).pa[31:3] \rangle]|_b & \text{otherwise} \end{cases}))
\end{aligned}
$$

We can now define a predicate for overall *MMU* correctness.

Definition 2.1.24 *We call a MMU specification trace correct iff it fulfills the following predicate:*

$$
\begin{aligned}
&mmu\_guarantee(trc) := \\
&\qquad\qquad p\_liveness(trc) \wedge \\
&\qquad\qquad m\_mr\_mw\_mutexc(trc) \wedge \\
&\qquad\qquad m\_req\_is\_stable(trc) \wedge \\
&\qquad\qquad untr\_read(trc) \wedge \\
&\qquad\qquad untr\_write(trc) \wedge \\
&\qquad\qquad tr\_read(trc) \wedge \\
&\qquad\qquad tr\_write(trc)
\end{aligned}
$$

We prove later for a *MMU* implementation that this predicate holds under the assumptions given earlier.

2.2 MMU Design

In this section we introduce an implementation of the *MMU*. We use only the basic circuits which were specified in Section 1.3.

Figure 2.5 shows the data paths of the *MMU*. In order to compute the *lexcp* we use the output *neg* from the circuit Add_sub_{21}. We compute a page table entry address with the help of the $Adder_{32}$. We save the data from the memory in the register *dr*. In the address register *ar* both a page table entry address or physical address can be stored. Since our memory supports only double word accesses and the page table entry is only one word wide we read

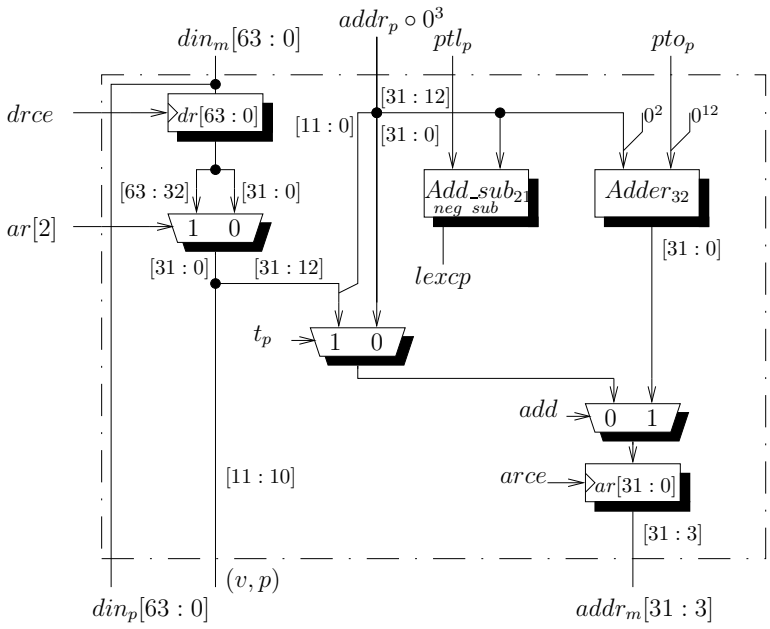

Figure 2.5: Data Paths of the *MMU*

two page table entries at the same time and by using $ar[2]$ we choose the right one. The last two multiplexers we use to compute the address which we store in the address register ar. The corresponding control automaton is presented in Figure 2.6.

The rest of the signals we define using interface signals and signals from Figure 2.5 in the following way:

$$\begin{aligned} mbw_m &= mbw_p \\ dout_m &= dout_p \\ pteexcp &= \neg v \vee p \wedge mw_p \\ excp_p &= lexcp \vee pteexcp \\ req_p &= mr_p \vee mw_p \\ busy_p &= \neg idle' \end{aligned}$$

where $idle'$ denotes that the control state in the next cycle will be *idle*.

Definition 2.2.1 *We introduce short notation s_x^t for $trc(t).iops_x.s$ where $t \in \mathbb{N}$ denotes a hardware cycle, x denotes one of two interfaces (p and m for the processor and memory interface correspondingly) and $trc \in Trace$, e.g. $trc(t).iobs_p.busy$ and $busy_p^t$ are the same for us.*

The next state function of the *MMU* implementation takes as inputs the following components:

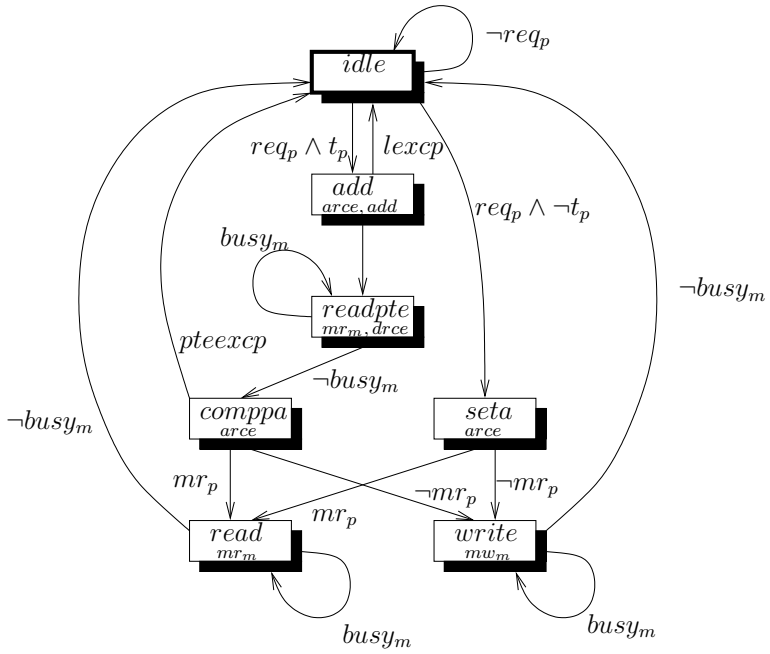

Figure 2.6: Control Automaton of the *MMU*

- $inputs_p^t$ – the inputs from the processor in current cycle t,
- $inputs_m^t$ – the inputs from the memory in current cycle t,
- c_{mmu}^t – configuration of the *MMU* in current cycle t. Configuration c_{mmu} contains the following components:
  - $c_{mmu}^t.ar$ – state of the address register ar in the current cycle t,
  - $c_{mmu}^t.dr$ – state of the data register dr in the current cycle t,
  - $c_{mmu}^t.st$ – state of the control automaton in the current cycle t.

and based on the data path and control automaton produces outputs as follows:

- $outputs_p^t$ – outputs to the processor in current cycle t,
- $outputs_m^t$ – outputs to the memory in current cycle t,
- c_{mmu}^{t+1} – configuration of the hardware in next cycle $t+1$

Components $inputs_p^t$, $inputs_m^t$, $outputs_p^t$, and $outputs_p^t$ induce a trace $trc \in Trace$. Note that initial state of the control automaton is $idle$, i.e. $c_{mmu}^0.st = idle$.

2.3 MMU Correctness

In this section we show the correctness of the *MMU*.

Let $trc \in Trace$ as before be a trace induced by the *MMU* implementation.

By correctness of the *MMU* we understand that if all the assumptions are fulfilled our implementation satisfies the guarantees. Formally, we have to prove the following lemma:

Lemma 2.3.1 *The implementation of the MMU satisfies the specification of the MMU in case the assumptions for both interfaces are fulfilled:*

$$good\_p\_interface(trc) \wedge good\_m\_interface(trc) \implies mmu\_guarantee(trc)$$

We need some intermediate lemmas in order to prove this claim. Now we start to prove the simple property from Definition 2.1.17 which guarantees that read and write signals in the memory are mutually exclusive.

Lemma 2.3.2 *It never holds that both signals write and read in the memory are active:*

$$m\_mr\_mw\_mutexc(trc)$$

Proof: This property directly follows from the control automaton: the memory read and memory write signals mr_m and mw_m are active in distinct states *read*, *readpte* for mr_m, and *write* for mw_m and thus we finish the proof. □

Now we want to prove the next property from Definition 2.1.18. At first we will prove only one part of this property for two inputs mr_m and mw_m. We start with the mr_m.

Lemma 2.3.3 *It always holds that if in the same cycle both signals mr_m and $busy_m$ are active in the next cycle mr_m is also active:*

$$\forall t \in \mathbb{N} : mr_m^t \wedge busy_m^t \implies mr_m^{t+1}$$

Proof: The proof of this lemma also follows from the inspection of the *MMU* control automaton: if mr_m is active we are in one of the two states *read* or *readpte*. The state does not change in the next cycle because $busy_m$ is active which proves that in the next cycle mr_m will also be active. □

In case of mw_m the lemma and its proof will be similar to the proof which we had in the previous lemma, we have only one difference: mw_m is active iff we stay in the state *write*.

Lemma 2.3.4 *The memory inputs are stable:*

$$p\_req\_is\_stable(trc) \implies m\_req\_is\_stable(trc)$$

Proof: By inspection of the control signals we can conclude that in case of a memory request in cycle t the control automaton is in one of the three states *readpte*, *read*, or *write*.

Since the control automaton is initially in state *idle*, we find a cycle t' before t when the processor request starts. The automaton leaves *idle* and all the time between time t' and time t, signal $busy_p$ is active. Processor inputs do not change between times t' and $t+1$ because of $p\_req\_is\_stable(trc)$ at cycle t. Thus, the data input into the memory and the memory byte write input into the memory (both of which come directly from the processor interface) do not change:

$$\begin{aligned} dout_m^t = dout_p^t &= dout_p^{t+1} = dout_m^{t+1} \\ mbw_m^t = mbw_p^t &= mbw_p^{t+1} = mbw_m^{t+1} \end{aligned}$$

The memory address is given by address register $ar[31:3]$ which is clocked only in states *seta* and *comppa*. Since we are neither in state *seta* or *comppa* in cycle t, we can conclude that:

$$addr_m^t = addr_m^{t+1}$$

For the last two signals mr_m and mw_m we have already proved this property before in Lemma 2.3.3. □

Lemma 2.3.5 *Processor liveness holds:*

$$m\_liveness(trc) \wedge p\_req\_is\_stable(trc) \implies p\_liveness(trc)$$

Proof: Since the definition of the busy signal for the processor interface is $p.busy = \neg idle'$ we can conclude that the processor interface is live if there is a later cycle $t' \geq t$ such that in cycle $t'+1$ the control enters the state *idle* again.

All cycles in the automaton either contain the state *idle* or are self-loops, i.e., we only have to prove that states in which we can have self-loops are eventually left. These states are *write*, *readpte*, and *write*. Assume we stay in one of these states in some cycle t''. Then we conclude that $mr_m^{t''} \vee mw_m^{t''}$ holds. From $m\_liveness(trc)$ in cycle t'' we conclude that $\exists t''' \in \mathbb{Z}_{\geq t''} : \neg busy_m^{t'''}$, i.e., we leave the state and thus, we reach state *idle* again. □

Now we want to prove the correctness of all memory operations requested by the processor. We start with the untranslated read request.

Lemma 2.3.6 *Any untranslated read request from the processor to the MMU is performed correctly:*

$$\begin{aligned} & p\_req\_is\_stable(trc) \wedge m\_liveness(trc) \wedge \\ & p\_mr\_mw\_mutexc(trc) \wedge m\_read\_consist(trc) \wedge \\ & \qquad \Longrightarrow untr\_read(trc) \end{aligned}$$

Proof: Let $t, t' \in \mathbb{N}$ satisfy $p\_is\_req(t, t', trc)$. Note that in both cycles t and $t'+1$ the control is in the *idle* state. In any intermediate cycle, it is not in the *idle* state. Note also that during the request all inputs are stable according to Lemma 2.1.8.

Since we have an untranslated request, i.e. $\neg t_p^t$, the control is in state *seta* in cycle $t+1$.

Since in cycle $t+1$ the control signal *arce* is active and the mux select signals *add* and t are both inactive, the address register $ar[31:0]$ is clocked as follows:

$$ar^{t+2}[31:0] = addr_p^{t+1} \circ 0^3 = addr_p^t \circ 0^3$$

Since we have a read request the control enters state *read* in cycle $t+2$ and we start the memory request. Because of $m\_liveness$ we know $\neg busy_m^{t''}$ for some $t'' \geq t+2$ and $busy_m^{t'''}$ for all t''' in between. With the help of Lemma 2.3.4 all memory inputs during a memory request are stable. Hence we conclude that we are in state *read* in cycle t'' and *idle* in cycle $t''+1$. This means that request ends in cycle t'', i.e., we have that $t'' = t'$. Based on the $m\_read\_consist$ assumption we conclude that:

$$\begin{aligned} din_m^{t'} &= trc(t').mem[\langle addr_m^{t'} \rangle] \\ &= trc(t').mem[\langle ar^{t'}[31:3] \rangle] = trc(t').mem[\langle ar^{t+2}[31:3] \rangle] \\ &= trc(t').mem[\langle addr_p^t \rangle] \end{aligned}$$

With the help of $m\_read\_consist$ we conclude that that the memory does not change between cycles t' and $t'+1$.

Thus, we obtain:

$$\begin{aligned} & din_p^{t'} = din_m^{t'} = trc(t').mem[\langle addr_p^t \rangle] \wedge \\ & trc(t').mem = trc(t'+1).mem \wedge \neg excp_p^{t'} \end{aligned}$$

□

Lemma 2.3.7 *Any untranslated write request from the processor to the MMU is performed correctly:*

$$\begin{aligned} & p\_req\_is\_stable(trc) \wedge m\_ack\_write(trc) \wedge \\ & p\_mr\_mw\_mutexc(trc) \wedge m\_write\_consist(trc) \wedge \\ & m\_liveness(trc) \implies untr\_write(trc) \end{aligned}$$

Proof: Let $t, t' \in \mathbb{N}$ satisfy $p\_is\_req(t, t', trc)$. As in the proof of Lemma 2.3.6 we conclude:

$$ar^{t+2}[31:0] = addr_p^{t+1} \circ 0^3 = addr_p^t \circ 0^3$$

In analogy to Lemma 2.3.6 we have $write^{t''}$ where $t+2 \leq t'' \leq t'$ and $\neg busy_m^{t'}$. Based on the $m\_write\_consist$ assumption we conclude that:

$$\begin{aligned}
&\forall a \in \mathbb{B}^{29} : (a \neq addr_m^{t+2} \implies \\
&\qquad trc(t').mem[\langle a \rangle] = trc(t'+1).mem[\langle a \rangle]) \wedge \\
&\forall b \in \mathbb{Z}_8 \left| trc(t'+1).mem[\langle addr_m^{t+2} \rangle] \right|_b = \\
&\qquad \begin{cases} \left| dout_m^{t+2} \right|_b & \text{if } mbw_m^{t+2}[b] \\ \left| trc(t').mem[\langle addr_m^{t+2} \rangle] \right|_b & \text{otherwise} \end{cases}
\end{aligned}$$

We also have the following equations from the construction of the *MMU*:

$$\begin{aligned}
dout_m^{t+2} &= dout_p^{t+2} \\
mbw_m^{t+2} &= mbw_p^{t+2} \\
addr_m^{t+2} &= ar^{t+2}[31:3]
\end{aligned}$$

We also do not have any cases of the exception by untranslated request. Thus, we obtain:

$$\begin{aligned}
&\forall a \in \mathbb{B}^{29} : (a \neq addr_p^t \implies \\
&\qquad trc(t').mem[\langle a \rangle] = trc(t'+1).mem[\langle a \rangle]) \wedge \\
&\neg excp_p^{t'} \wedge \\
&\forall b \in \mathbb{Z}_8 \left| trc(t'+1).mem[\langle addr_p^t \rangle] \right|_b = \\
&\qquad \begin{cases} \left| dout_p^t \right|_b & \text{if } mbw_p^t[b] \\ \left| trc(t').mem[\langle addr_p^t \rangle] \right|_b & \text{otherwise} \end{cases}
\end{aligned}$$

□

Lemma 2.3.8 *Any translated read request from the processor to the MMU is consistent:*

$$\begin{aligned}
&p_req_is_stable(trc) \wedge m_ack_write(trc) \wedge \\
&p_mr_mw_mutexc(trc) \wedge m_read_consist(trc) \wedge \\
&m_liveness(trc) \implies tr_read(trc)
\end{aligned}$$

Proof: Let $t, t' \in \mathbb{N}$ be the start and end time for any translated processor read request. The control is in state *idle* at cycles t and $t'+1$. In any cycle of the intermediate cycles, it is not in state *idle*.

Since t_p^t holds, the control is in state *add* in cycle $t+1$. Since $arce^{t+1}$ and the mux select signals add^{t+1} and t_p^{t+1} are active, the address register ar is clocked as follows:

$$ar^{t+2}[31:0] = adder_{32}(0^{10} \circ addr_p^{t+1}[28:9] \circ 0^2, pto_p^{t+1} \circ 0^{12}, 0)[31:0]$$

By using Definition 1.3.1 of an *adder* and input stability we can rewrite this equation as follows:

$$\langle ar^{t+2}[31:0] \rangle = \langle addr_p^t[28:9] \circ 0^2 \rangle + \langle pto_p^t \circ 0^{12} \rangle \bmod 2^{32} = r\_t(t).ptea$$

That proves, that we have saved the page table entry address $r\_t(t).ptea$ in the address register at cycle $t+2$.

We compute $lexcp^{t+1}$ as $add\_sub_{21}(0 \circ ptl_p^{t+1}, 0 \circ addr_p^{t+1}[28:9], 1).neg$ and by Definition 1.3.1 we have

$$lexcp^{t+1} = (\langle ptl_p^t \rangle - \langle addr_p^t[28:9] \rangle < 0)$$

i.e. $lexcp^{t+1} = r\_t(t).lexcp$

We now split cases on $lexcp^{t+1}$:

1. Let $lexcp^{t+1}$ hold. Of course, then the control is in state *idle* at cycle $t+2$ and we have $t' = t+1$. Because of $lexcp^{t+1}$, we have $r\_t(t).excp$. This concludes the case since we do not change the memory.

2. Let $\neg lexcp^{t+1}$ hold. Then the control is in the state *readpte* in cycle $t+2$. By using the $m\_liveness$ assumption we know that at a later time $t'' \geq t+2$, we have $\neg busy_m^{t''}$ holds, and at this time we save the input data from the memory $din_m^{t''}$ into the data register dr. The input data from the memory at the time t'' contains two adjacent page table entries whose address differs in the last bit. With the help of mux select signal $ar[2]$ we later choose the page table entry which is needed. Since the memory request at address $r\_t(t).ptea$ in cycle $t''+1$ is finished we also know that

$$r\_t(t).pte = \begin{cases} dr^{t''+1}[31:0] & \text{if } \langle r\_t(t).ptea \rangle \bmod 8 = 0 \\ dr^{t''+1}[63:32] & \text{otherwise} \end{cases}$$

   The control is at cycle $t''+1$ in state *comppa*. Since the address register ar is not changed between cycles $t+2$ and $t''+1$ we have $r\_t(t).ptea = ar^{t''+1}$. Since $r\_t(t).ptea[1:0] = 0^2$ and processor inputs are stable we have:

$$\langle r\_t(t).ptea \rangle \bmod 8 = 0 \iff ar^{t''+1}[2]$$

Since in cycle $t''+1$ the control signal $arce^{t''+1}$ and the mux select signal $t_p^{t''+1}$ are active and the mux select signal $add^{t''+1}$ is inactive, the address register $ar^{t''+2}[31:0]$ is given by:

$$\begin{aligned} ar^{t''+2}[31:0] &= r\_t(t).pte[31:12] \circ addr_p^{t''+1}[8:0] \circ 0^2 \\ &= r\_t(t).pte[31:12] \circ bx \circ 0^2 \end{aligned}$$

We compute the $pteexcp^{t'}$ as $\neg v \vee p \wedge mw_p^t = \neg r\_t(t).pte[11] \vee r\_t(t).pte[10] \wedge mw_p^t$. Since we do not have mw_p^t because of read request we can rewrite the $pteexcp^{t'}$ just as $\neg r\_t(t).pte[11]$.

We will now split cases on $pteexcp^{t''+1}$:

(a) Let $pteexcp^{t''+1}$ hold. Of course, then the control is in the state *idle* at cycle $t''+2$ and we obtain that $t''+1=t'$. Note that we do not change the data register during cycle $t''+1$ and so we have $r\_t(t).excp$. This concludes the claim since we do not change the memory during the whole request.

(b) If $\neg pteexcp^{t''+1}$ holds then we have $\neg r\_t(t).excp$ and $r\_t(t).pa = ar^{t''+2}[31:3]$. Since we have a read request the control is in the state *read* at cycle $t''+2$ we start the second memory read request at the physical address stored in ar. With the help of the $m\_liveness$ assumption we know that $\neg busy_m^{\tilde{t}}$ for a later time $\tilde{t} \geq t''+2$ and all the time between $t''+2$ and $\tilde{t}$ the memory is busy. This means that $\tilde{t}=t'$. Now based on the $m\_read\_consist$ assumption we can conclude that:

$$\begin{aligned} din_m^{t'} &= trc(t').mem[\langle addr_m^{t'} \rangle] \\ &= trc(t').mem[\langle ar^{t'}[31:3] \rangle] \\ &= trc(t').mem[\langle ar^{t''+2}[31:3] \rangle] \\ &= trc(t').mem[r\_t(t).pa] \end{aligned}$$

This finishes the proof since during the processor request the memory is not changed. □

Lemma 2.3.9 *Any translated write request from the processor to the MMU is consistent:*

$$\begin{aligned} &p\_req\_is\_stable(trc) \wedge m\_read\_consist(trc) \wedge \\ &p\_mr\_mw\_mutexc(trc) \wedge m\_write\_consist(trc) \wedge \\ &m\_liveness(trc) \wedge m\_ack\_write(trc) \implies \\ &\qquad tr\_write(trc) \end{aligned}$$

Proof: The proof of this lemma is very similar to the previous one. We only have two differences:

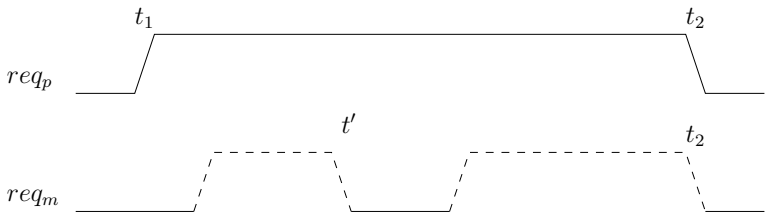

Figure 2.7: Possibility of Requests to the Memory

1. The signal *pteexcp* can now also be active because the protection bit is set. The correctness of *pteexcp* is concluded similar to the *pteexcp* from Lemma 2.3.8.

2. The second memory access now is a write access instead of a read access. At the end of the request the control is in the state *write* and we store the data into the memory according to the definition of untranslated write. The correctness of this write is concluded similar to the untranslated write from Lemma 2.3.7. □

We can now summarize all of the lemmas from 2.3.2 to 2.3.9 to prove Lemma 2.3.1.

Proof: [Lemma 2.3.1] We have already proved all parts of the predicate $mmu\_guarantee(trc)$ in the lemmas from 2.3.2 to 2.3.9. Both predicates $good\_p\_interface(trc)$ and $good\_m\_interface(trc)$ also contain all the assumption which we needed for these lemmas. □

We now have the correctness proof for the *MMU* but in order to use this proof for the overall proof for the instruction *MMU* of the VAMP we need to have one more lemma. The assumption that a *MMU* has exclusive access to the memory does not hold in real processors (cf. Chapter 3). Therefore we need the following lemma in order to distinguish memory space for different *MMU*s.

If we have a memory request which ends before end of the processor request then we have a translated operation and the address of this request is $\lfloor \langle r\_t(t).ptea \rangle / 8 \rfloor$. If we also have a request to the memory at the end of the processor request then the address of this request is equal to $r\_t(t).pa$ (see Figure 2.7).

Lemma 2.3.10 *In case of a processor read request we have the same addresses on the address bus of the memory interface at the end of the memory*

request as in specification of the MMU:

$$\begin{aligned}
&\forall t_1, t_2 \in \mathbb{N} : is_req_proc(t_1, t_2, trc) \wedge \\
&\quad p_mr_mw_mutexc(trc) \wedge p_req_is_stable(trc) \implies \\
&\quad (mr_p^{t_1} \implies (\forall t' \in [t_1 : t_2[\ : \ (mr_m^{t'} \vee mw_m^{t'}) \wedge \neg busy_m^{t'} \implies \\
&\qquad\qquad \langle addr_m^{t'} \rangle = \lfloor \langle r_t(t).ptea \rangle / 8 \rfloor \wedge t_p^{t_1})) \\
&\quad \wedge \\
&\quad ((mr_m^{t_2} \vee mw_m^{t_2}) \wedge \neg busy_m^{t_2}) \implies \\
&\qquad \begin{cases} \exists t' \in [t_1 : t_2[\ : \ (mr_m^{t'} \vee mw_m^{t'}) \wedge \neg busy_m^{t'} \wedge & \\ \quad addr_m^{t_2} = r_t(t).pa[31:3] & \text{if } t_p^{t_2} \\ addr_m^{t_2} = addr_p^{t_2} & \text{otherwise} \end{cases}
\end{aligned}$$

Proof: We prove the two implications of the claim separately.

1. First we show that:

$$\begin{aligned}
&\forall t' \in [t_1 : t_2[: (mr_m^{t'} \vee mw_m^{t'}) \wedge \neg busy_m^{t'} \implies \\
&\qquad\qquad \langle addr_m^{t'} \rangle = \lfloor \langle r_t(t_1).ptea \rangle / 8 \rfloor \wedge t_p^{t_1}
\end{aligned}$$

   From $mr_m^{t'} \vee mw_m^{t'}$ we conclude that the automaton is at cycle t' in one of the tree states: *read*, *write*, or *readpte*. Moreover, if the control is in states *read* or *write* and t' is a last cycle of memory request then follows that $t' = t_2$ which is a contradiction because $t' < t_2$. Therefore the control is in state *readpte* in cycle t'. We can only stay in state *readpte* when we have a translated request, i.e. the signal t_p is active during the processor request.

   This concludes the claim since

$$addr_m^{t'} = ar[31:3] = \lfloor r\_t(t_1).ptea/8 \rfloor$$

   we have already shown as part of Lemma 2.3.8.

2. Now we show that:

$$\begin{aligned}
&((mr_m^{t_2} \vee mw_m^{t_2}) \wedge \neg busy_m^{t_2}) \implies \\
&\quad \begin{cases} \exists t' \in [t_1 : t_2[\ : \ (mr_m^{t'} \vee mw_m^{t'}) \wedge \neg busy_m^{t'} \wedge & \\ \quad addr_m^{t_2} = r_t(t).pa[31:3] & \text{if } t_p^{t_2} \\ addr_m^{t_2} = addr_p^{t_2} & \text{otherwise} \end{cases}
\end{aligned}$$

   We have already proved this case when $t_p^{t_2}$ holds as part of Lemma 2.3.8 and other case we have proved as part of Lemma 2.3.6 and thus we finish the claim. □

Chapter 3

The VAMP with Virtual Memory Support

The work presented in this chapter is based on the works of Sven Beyer [Bey05], Mark Hillebrand [Hil05], and Daniel Kröning [Krö01].

The processor which we describe in this chapter has two principle differences with respect to the processor in [Bey05]. First we will have an implementation supporting *virtual memory* which allows a user program to use more memory than the physical RAM available. Second we add a new type of interrupts for communication with external environment.

3.1 Specification of the VAMP

Typically, the specification of a microprocessor consists of three components: a memory, a register file and a program counter. A step of computation is the execution of one command which is fetched from the memory at the address given by the program counter. Our specification follows this scheme.

In our case we have three register files:

- $GPR : \{0,1\}^5 \rightarrow \{0,1\}^{32}$ – a general purpose register file. It consists of 32 registers. Every register has a 32-bit width. The $GPR[0]$ always contains 0^{32}.
- $FPR : \{0,1\}^5 \rightarrow \{0,1\}^{32}$ – a floating point register file. It consists of 32 registers. Every register has a 32-bit width. All registers in the floating point register file also can be used in even-odd pairs, encoding an IEEE double-precision floating point number.
- $SPR : \mathbb{Z}_{17} \rightarrow \{0,1\}^{32}$ – a special purpose register file. It consists of 17 registers. Every register has a 32-bit width. We will discuss the purpose of these registers later on. For now only note that we have

four new registers in the *SPR*, which are needed for address translation. They are *PTO*, *PTL*, *MODE* and *EMODE*.

For the program counter we use a so-called *delayed PC* construction. In such a delayed PC architecture, instruction updates to the PC do not effect the next instruction. We need two program counters - $PC' \in \{0,1\}^{32}$ and $DPC \in \{0,1\}^{32}$. Full definition of this construction is provided by Müller and Paul [MP00]. Therefore, the VAMP specification configuration consist of six components

$$c_S = (c_S.GPR, c_S.SPR, c_S.FPR, c_S.M, c_S.PC', c_S.DPC) \tag{3.1}$$

Note that $c_S.M \in Memory$. Let us introduce some notations, which will be used in this chapter. We will consider two different specification configuration. The first is configuration c_S which reacts on interrupts and the second is $\tilde{c}_S$ which does not react on interrupts. We denote with c_S^n our specification configuration before the execution of instruction n and after executing instruction $n-1$. We also denote with $\tilde{c}_S^n$ our specification configuration in the case of not having interrupts. We also introduce two next step functions δ and δ_u which reacts and does not react on interrupts correspondingly. The first next step function takes as input only specification configuration. We now define a computation without interrupts with an initial configuration c_S^{init} as follows:

$$\begin{aligned} \tilde{c}_S^0 &:= c_S^{init} \\ \tilde{c}_S^{n+1} &:= \delta_u(\tilde{c}_S^n). \end{aligned}$$

The second function takes additionally external signal *reset* and since we add external interrupts $ext[18:0]$ takes this bitvector as input. Therefore a computation with interrupts with an initial computation c_S^{init} is defined as follows:

$$\begin{aligned} c_S^0 &:= c_S^{init} \\ c_S^{n+1} &:= \delta(c_S^n, reset, ext_S[18:0]). \end{aligned}$$

We denote with $c_S^n.F$ one of the six components in the configuration before the execution of instruction n.

At first we will develop the next step computation for $\tilde{c}_S^{n+1} = \delta_u(\tilde{c}_S^n)$ (or c'_S) and then we will extend it to the $c_S^{n+1} = \delta(c_S^n, reset, ext[18:0])$.

Our processor supports two modes of operation. The first mode is called *system mode*. In this mode we do not have *virtual addresses*, we have only *physical addresses*. The behavior of our processor in this mode is equal to the behavior of the processor described in [Bey05].

The second mode is called *user mode*. In this mode we have *virtual addresses* which we translate into *physical addresses* when we want to fetch

from, read from, or write to the memory. The current mode is indicated by the lowest bit in the *MODE* register. If this bit equals 0 then we are in *system mode*, otherwise in *user mode*. We denote this bit by t, i.e. $t(c_S) = c_S.SPR[MODE][0]$. Note that all names of registers in the special purpose register file are natural numbers and are equal to numbers from the first column in Table 3.1.

For address translation we will use the *decodeitr* function which was defined in Section 2.1 on the MMU specification. Based on the *decodeitr* function for memory access to the instruction memory we define instruction page fault $ipf(c_S)$ and instruction physical address $ipa(c_S)$:

$$ipf(c_S) = decodeitr(pto, ptl, c_S.M, 0, c_S.DPC[31:3]).excp, \quad (3.2)$$

$$ipa(c_S) = decodeitr(pto, ptl, c_S.M, 0, c_S.DPC[31:3]).pa \quad (3.3)$$

where *pto* and *ptl* are defined as

$$pto = c_S.SPR[PTO][19:0], \quad (3.4)$$

$$ptl = c_S.SPR[PTL][19:0] \quad (3.5)$$

Every fetch access to the memory must not have *instruction misalignment*, i.e. must be aligned, therefore for the access to the instruction memory we define a predicate

$$imal(c_S) = c_S.DPC \bmod 4 \neq 0 \quad (3.6)$$

The function *IR* returns the current instruction, which we want to execute in configuration c_S. We define it as follows:

$$IR(c_S) = \begin{cases} c_S.M[c_S.DPC + 3 : c_S.DPC] & \text{if } \neg t \wedge \neg imal(c_S) \\ c_S.M[ipa(c_S) + 3 : ipa(c_S)] & \text{if } t \wedge \neg imal(c_S) \wedge \neg ipf(c_S) \\ 0^{32} & \text{otherwise} \end{cases} \quad (3.7)$$

An instruction consist of several fields, which encode whole instruction set of the processor according to the Tables A.1 to A.6.

Based on $IR(c_S)$ and Figure A.1 in Appendix A, we define some functions. For example $RS1(c_S)$ contains the index of the first source register in a register file, $RD(c_S)$ contains the index of the destination register in a register file, $SA(c_S)$ contains the index of the register in the *SPR* register file (source or destination register depending on the command), etc.

Based on the Tables A.1 to A.6 we also introduce some predicates. For example `movs2i`?(IR) holds iff $IR[31:26] = 0^6 \wedge IR[5:0] = 010^4$ (Table A.2). Also we have predicates identifying groups of instructions, e.g., `mem`?(c_S) is true for all memory instructions.

Based on the *decodeitr* function in case if predicate `mem`? holds we define data page fault $dpf(c_S)$ and data physical address $dpa(c_S)$. As *virtual address* we have effective memory address:

$$ea(c_S) := c_S.GPR[RS1] + imm(c_S) \quad (3.8)$$

According to the Tables A.1 to A.6 we have memory write access only in the case of:

$$mw := \texttt{sb}? \vee \texttt{sh}? \vee \texttt{sw}? \vee \texttt{store.s}? \vee \texttt{store.d}? \tag{3.9}$$

We define physical address and page fault for memory data access as follows:

$$dpf(c_S) = decodeitr(pto, ptl, c_S.M, mw, ea(c_S)).excp, \tag{3.10}$$
$$dpa(c_S) = decodeitr(pto, ptl, c_S.M, mw, ea(c_S)).pa \tag{3.11}$$

The *pto* and *ptl* are the same as in the case of instruction memory access. Note that data access must also be aligned:

$$dmal(c_S) = \begin{cases} ea(c_S) \bmod d \neq 0 & \text{if } \neg t \wedge \texttt{mem}? \\ dpa(c_S) \bmod d \neq 0 & \text{if } t \wedge \texttt{mem}? \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

where d is the width of the access measured in bytes as introduced in Tables A.1 and A.4.

Now we can define the next step computation for the memory:

$$c'_S.M = \begin{cases} c_S.M \; with \; c_S.M[ea(c_S) + d - 1 : ea(c_S)] := dest[8 \cdot d - 1 : 0] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } \neg t \wedge \neg dmal(c_S) \wedge mw \\ c_S.M \; with \; c_S.M[dpa(c_S) + d - 1 : dpa(c_S)] := dest[8 \cdot d - 1 : 0] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{if } t \wedge \neg dmal(c_S) \wedge \neg dpf(c_S) \wedge mw \\ c_S.M \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

where

$$dest = \begin{cases} c_S.GPR[RD(c_S)] & \text{if } \texttt{sb}? \vee \texttt{sh}? \vee \texttt{sw}? \\ c_S.FPR[RD(c_S)] & \text{if } \texttt{store.s}? \\ c_S.FPR[RD(c_S)[4:1]1] \circ c_S.FPR[RD(c_S)[4:1]0] & \text{otherwise} \end{cases}$$

Then the next step computation for the *GPR* is defined as follows:

$c'_S.GPR[RD(c_S)] =$

$$\begin{cases} sext(c_S.M[ea(c_S) : ea(c_S)]) & \text{if } \texttt{lb}? \wedge \neg dmal(c_S) \wedge \neg t \\ sext(c_S.M[dpa(c_S) : dpa(c_S)]) & \text{if } \texttt{lb}? \wedge \neg dmal(c_S) \wedge \\ & \qquad t \wedge \neg dpf(c_S) \\ sext(c_S.M[ea(c_S)+1 : ea(c_S)]) & \text{if } \texttt{lh}? \wedge \neg dmal(c_S) \wedge \neg t \\ sext(c_S.M[dpa(c_S)+1 : dpa(c_S)]) & \text{if } \texttt{lh}? \wedge \neg dmal(c_S) \wedge \\ & \qquad t \wedge \neg dpf(c_S) \\ c_S.M[ea(c_S)+3 : ea(c_S)] & \text{if } \texttt{lw}? \wedge \neg dmal(c_S) \wedge \neg t \\ c_S.M[dpa(c_S)+3 : dpa(c_S)] & \text{if } \texttt{lw}? \wedge \neg dmal(c_S) \wedge \\ & \qquad t \wedge \neg dpf(c_S) \\ c_S.GPR[RS1(c_S)] \wedge imm(c_S) & \text{if } \texttt{andi}? \\ c_S.GPR[RS1(c_S)] - imm(c_S) & \text{if } \texttt{subi}? \\ \quad \vdots & \qquad \vdots \\ c_S.GPR[RD(c_S)] & \text{otherwise} \end{cases}$$

Note that with *sext* we denote the operation which we use in order to produce the sign extension result. We can describe the behavior of both program counters in the following way:

$$c'_S.PC' = \begin{cases} c_S.PC' + 4 + imm(c_S) & \text{if } \texttt{beqz}? \wedge RS1(c_S) = 0 \vee \\ & \quad \texttt{bnez}? \wedge RS1(c_S) \neq 0 \vee \texttt{j}? \vee \texttt{jal}? \\ c_S.GPR[RS1(c_S)] & \text{if } \texttt{jr}? \vee \texttt{jalr}? \\ c_S.PC' + 4 & \text{otherwise} \end{cases} \tag{3.13}$$

$$c'_S.DPC = c_S.PC' \tag{3.14}$$

We will not formally define the next step function for the *FPR* register file. All details about *FPR* can be found in [Jac02a] and in [Bey05]. For the *SPR* we first describe the next step computation without interrupts for the *IEEEf* register. The *IEEEf* register saves the exception which we needed according to IEEE standard for floating-point operations. We compute this register as follows:

$c'_S.SPR[IEEEf] =$

$$\begin{cases} c_S.GPR[RS1(c_S)] & \text{if } \texttt{movi2s}? \wedge \\ & SA = IEEEf \\ c_S.SPR[IEEEf][31:5](c_S.SPR[IEEEf][4:0] \vee & \text{otherwise} \\ \qquad CA(c_S)[11:7]) & \end{cases}$$

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.
9	*PTO*	Page table origin.
10	*PTL*	Page table length.
11	*EMODE*	Exception *MODE*. Saves *MODE* in case of an interrupt.
12	*S*12	Not used for special purpose.
13	*S*13	Not used for special purpose.
14	*S*14	Not used for special purpose.
15	*S*15	Not used for special purpose.
16	*MODE*	Used to distinguish *system* and *user* mode.

Table 3.1: Special Purpose Registers of the VAMP

Note that definition for $CA(c_S)$ we will be described later. More information about this case you can find in [Bey05].

For all other registers in the *SPR* file we introduce the next step computation without interrupts as follows:

$$c'_S.SPR = \lambda_{i \in \mathbb{Z}_{17} \wedge i \neq 7} \begin{cases} c_S.GPR[RS1(c_S)] & \text{if } \texttt{movi2s?} \wedge SA(c_S) = i \\ c_S.SPR[i] & \text{otherwise} \end{cases} \tag{3.15}$$

Now we want to define specification computations with interrupts. We now describe the *SPR* register file. The register file contains 17 registers, seven registers dealing with interrupts, three for floating point operations and the new group of registers which we use for address translation. The purpose of each register is presented in the Table 3.1.

Note that we have four registers in the special purpose register file which are not used for any special purpose, i.e. they are only read or updated using

`movs2i`?(*IR*) or `movi2s`?(*IR*) instructions. This has implementation-specific reasons, which are described in Section 3.2.2.

All supported interrupts are given in Table 3.2. When an interrupt occurs we execute so-called jump to the interrupt service routine which starts at a fixed address *SISR*. We have two different kinds of interrupts: repeat and continue. If we have a repeat interrupt then after executing the interrupt service routine we start again to execute the instruction which caused the interrupt. Otherwise after executing the interrupt service routine we execute the next instruction after the instruction which caused the interrupt. We can also classify interrupts as maskable, i.e., they can be ignored under software control, and nonmaskable. We save the mask of interrupts in the *SR* register. When we have more than one interrupt simultaneously, the software executes an interrupt with the smallest index. For the specification of interrupts we define two additional functions, i.e., $CA(c_S)$ for cause exception (output of this function is a bit vector and the value of each bit indicates whether we have an exception of this index or not) and $EData(c_S)$ for exception data of the specification configuration. Our processor also has external interrupt $reset$ and 19 external interrupts for the communication with external devices.

We have the illegal exception $CA(c_S)[ill]$ in the following cases:

- if we cannot decode the new instruction in IR_S as one of the instructions according to Tables A.1 to A.5. This case also is in the previous VAMP.
- If user processes were allowed to update certain registers they could hack the operating system. Therefore we have to forbid any access to special purpose registers except *IEEEf*, *SR*, RM, FCC in $user$ mode. That is we have an illegal interrupt if we decode the new instruction in $IR(c_S)$ as access to *SPR* register file (i.e., `movi2s` or `movs2i`) but not to *IEEEf*, *SR*, RM, FCC registers and we work in $user$ mode.
- For the same reason we have to forbid any `rfe` command in $user$ mode i.e. if we decode the new instruction as `rfe` and we work in $user$ mode.

For misalignment exception we will have the following:

$$CA(c_S)[mal] = imal(c_S) \vee dmal(c_S) \tag{3.16}$$

For page fault on fetch and page fault on load / store we will have the following:

$$CA(c_S)[ipf] = ipf(c_S) \wedge \neg imal(c_S)$$
$$CA(c_S)[dpf] = dpf(c_S) \wedge \texttt{mem?} \wedge \neg dmal(c_S)$$

$CA(c_S)[trap]$ is true when we execute a `trap` instruction. $CA(c_S)[ovf]$ is true when we have overflows by fixed point arithmetic and we execute

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
13-31	*ext*[*i*]	continue	yes	19 external interrupts

Table 3.2: Supported Interrupts in the VAMP

an instruction that detects overflow. All other exceptions are used only for floating point arithmetic and we do not consider these exceptions here.

We compute the exception data in the following way:

$$EData(c_S) := \begin{cases} DPC(c_S) & \text{if } imal(c_S) \vee ipf(c_S) \\ sext(imm(c_S)) & \text{if } \texttt{trap}? \\ ea(c_S) & \text{if } \texttt{mem}? \\ FPUresult_S & \text{if } \texttt{fpu}? \\ 0^{32} & \text{otherwise} \end{cases} \tag{3.17}$$

Note that we need to save not only the exception data in the case of an interrupt but also the bit t, i.e. the register $c_S.SPR[MODE]$. For this purpose we extended the *SPR* register file with an *EMODE* register.

Now for the type of the interrupt we have:

$$repeat(c_S) := \bigvee_{i=0}^{i<5} CA(c_S)[i]. \tag{3.18}$$

For detecting an interrupt we define the masked cause function as follows:

$$MCA(c_S) := \lambda_{i \in \mathbb{Z}_{32}} \begin{cases} CA(c_S)[i] \wedge c_S.SPR[SR][i] & \text{if } i \geq 6 \wedge i \neq 12 \\ CA(c_S)[i] & \text{otherwise} \end{cases} \tag{3.19}$$

Note that equation $i \geq 6 \wedge i \neq 12$ holds only for the maskable interrupts. Now we can define the predicate $JISR(c_S)$ which indicates if any interrupt occurs:

$$JISR(c_S) := MCA(c_S) \neq 0^{32} \tag{3.20}$$

Note that when $\mathit{JISR}(c_S)$ occurs we jump into the interrupt service routine at address *SISR*. Computation steps with interrupts are equal to computation steps without interrupts when we do not have $\mathit{JISR}(c_S)$ and our specifications are always equal if we do not have any interrupts. We define our next step computation function in the following way.

For both program counters as:

$$c_S^{n+1}.\mathit{DPC} := \begin{cases} \mathit{SISR} & \text{if } \mathit{JISR}(c_S^n) \\ c_S^{n\prime}.\mathit{DPC} & \text{otherwise} \end{cases} \tag{3.21}$$

$$c_S^{n+1}.\mathit{PC}' := \begin{cases} \mathit{SISR}+4 & \text{if } \mathit{JISR}(c_S^n) \\ c_S^{n\prime}.\mathit{PC}' & \text{otherwise} \end{cases} \tag{3.22}$$

For the memory we have the following:

$$c_S^{n+1}.M = \begin{cases} \mathit{repeat}(c_S^n)?\, c_S^n.M : c_S^{n\prime}.M & \text{if } \mathit{JISR}(c_S^n) \\ c_S^{n\prime}.M & \text{otherwise} \end{cases} \tag{3.23}$$

For the *GPR* we have the following:

$$c_S^{n+1}.\mathit{GPR} = \begin{cases} \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{GPR} : c_S^{n\prime}.\mathit{GPR} & \text{if } \mathit{JISR}(c_S^n) \\ c_S^{n\prime}.\mathit{GPR} & \text{otherwise} \end{cases} \tag{3.24}$$

For the *FPR* we have the same construction as for *GPR* register file.

The next step computation with interrupts c_S^{n+1} for the *SPR* file is equal to the next step computation without interrupts $c_S^{n\prime}$ when we do not have $\mathit{JISR}(c_S^n)$. In the case of $\mathit{JISR}(c_S^n)$ we compute c_S^{n+1} for the *SPR* in the following way:

$$c_S^{n+1}.\mathit{SPR} =$$

$$\lambda_{i\in\mathbb{Z}_{17}} \begin{cases} 0^{32} & i = \mathit{SR} \\ \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{SPR}[\mathit{SR}] : c_S^{n\prime}.\mathit{SPR}[\mathit{SR}] & i = \mathit{ESR} \\ \mathit{MCA}(c_S^n) & i = \mathit{ECA} \\ \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{PC}' : c_S^{n\prime}.\mathit{PC}' & i = \mathit{EPC} \\ \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{DPC} : c_S^{n\prime}.\mathit{DPC} & i = \mathit{EDPC} \\ \mathit{EData}(c_S^n) & i = \mathit{EData} \\ \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{SPR}[\mathit{MODE}] : c_S^{n\prime}.\mathit{SPR}[\mathit{MODE}] & i = \mathit{EMODE} \\ \mathit{repeat}(c_S^n)?\, c_S^n.\mathit{SPR}[i] : c_S^{n\prime}.\mathit{SPR}[i] & \text{otherwise} \end{cases}$$

After executing an interrupt service routine we need to restore all necessary registers. Therefore the last command in the interrupt service routine

should be the return from the exception command `rfe`. This instruction changes PC', *DPC* and *SPR* in the following way:

$$\begin{aligned} c_S^{n+1}.PC' &:= c_S^n.SPR[EPC] \\ c_S^{n+1}.DPC &:= c_S^n.SPR[EDPC] \\ c_S^{n+1}.SPR[SR] &:= c_S^n.SPR[ESR] \\ c_S^{n+1}.SPR[MODE] &:= c_S^n.SPR[EMODE] \end{aligned}$$

This completes the formal definition of the specification of the microprocessor with MMU. Later we introduce a implementation and prove that our implementation realizes this specification.

In addition the following proposition trivially holds.

Proposition* 3.1.1 *As long as no interrupt occurs, both specification computations are equal, i.e., for all* $n \in \mathbb{N}$*:*

$$(\forall m \in \mathbb{Z}_n : \neg JISR(\tilde{c}_S^m)) \implies \tilde{c}_S^n = c_S^n$$

Note that we also can exchange $JISR(\tilde{c}_S^m)$ with $JISR(c_S^m)$.

3.2 Implementation of the VAMP

3.2.1 Tomasulo Algorithm

There are two principles of instruction execution in the microprocessors. The first is called in-order. In this case all the instructions are executed in the pipeline consecutively, i.e. instruction I_i will be completely executed after instruction I_{i-1} and before instruction I_{i+1} even if I_i needs no result of I_{i-1}. The second approach is called out-of-order (OOO) execution. In this case an instruction I_i that does not need any input from the instruction I_{i-1} can be executed before or simultaneously with the instruction I_{i-1}.

The implementation of the VAMP processor is based on the out-of-order implementation principle. We implement the so-called Tomasulo algorithm [Tom67]. The Tomasulo algorithm was first implemented in the IBM 360/91 Floating Point Unit and is used in many modern processors, e.g. Pentium 2, 3, 4; AMD K5, K6 and Athlon; Power P6, 603/ 604/ G3/ G4; MIPS R10000, R12000; Alpha 21264.

In order to realize precise interrupts we implemented this algorithm with a so-called *reorder buffer* (*ROB*) [SP88]. We call an interrupt between instructions I_{i-1} and I_i precise iff instructions $I_0, \ldots, I_{i-1}$ were completed before starting the interrupt service routine (*ISR*) and the processor did not change the state of the machine for instructions $I_i, I_{i+1}, \ldots$. With the help of the *ROB* we execute instructions out-of-order inside the pipeline, while

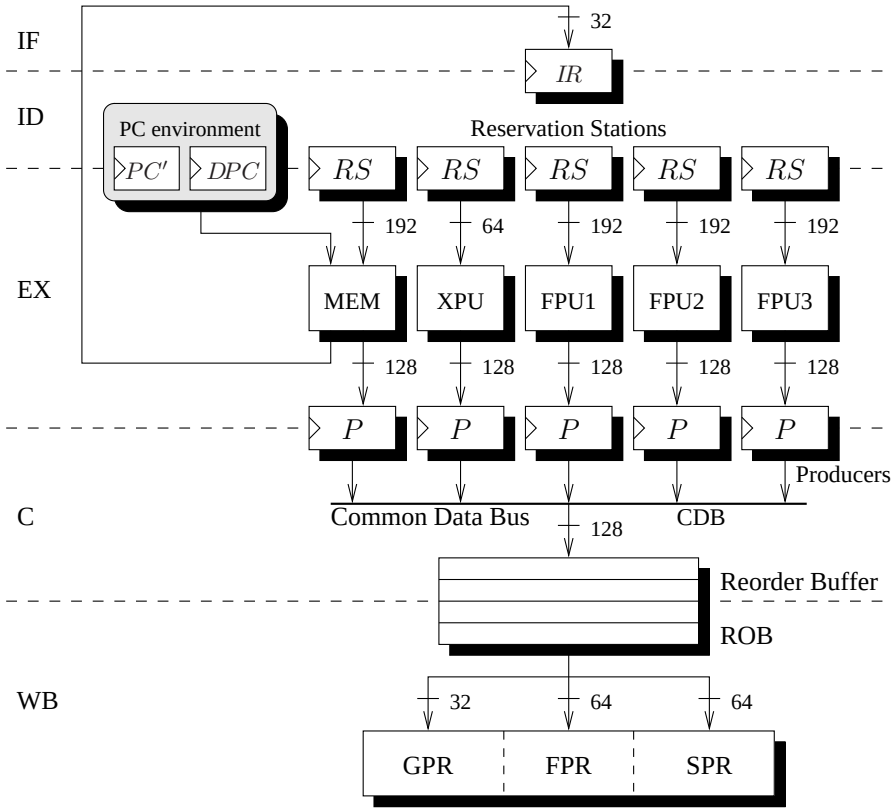

Figure 3.1: The VAMP Data Paths

the instructions leave the pipeline in-order. We also extend the register file with a so-called *producer table* in order to implement OOO execution. The additional information per register is a valid bit and a tag. If the valid bit is active then no instruction in the pipeline writes to the corresponding register, i.e. the register contains the correct data. If the valid bit is inactive we know that an instruction in the pipeline writes into this register. The tag is assigned during instruction issue. It stays unique until the instruction terminates.

Figure 3.1 shows the data paths of the VAMP processor. The execution begins with the instruction fetch. The Tomasulo scheduling algorithm does not cover this phase. In this phase the instruction is loaded from the memory. The next stage is ID. In this stage the instruction is decoded and issued to a *reservation station* (*RS*) with its source operands. Each *functional unit* (FU) has its own reservation stations. We also reserve a new *tag* for this instruction and the destination registers of this instruction we mark as invalid. We set the tag of each destination register to the instruction's *tag*. If not all operands are valid in registers or present in *ROB* we need to save for each unavailable source operand a tag of the instruction which must return

this operand as a destination register. In order to find unavailable operands on the *RS* we watch on the *Common Data Bus* (*CDB*), which we describe later, for the operands which are missing. This it is called *CDB snooping*. When all operands are valid and the *FU* also will be free we execute this instruction. After the execution we save execution result in the *producer* (P). After that the producer puts the result on the *CDB* the result of this instruction is available for all instructions which wait for it. We also save the result in the *ROB*. During the execution of stages EX and WB of pipeline this instruction is invalid in *ROB*. As soon as the oldest result in the *ROB* is valid we can write the result from *ROB* into register files. Note that we have an in-order output from *ROB* because we also add the instruction in *ROB* in-order during the instruction issue. Additional details about the Tomasulo algorithm with reorder buffer can be found in the thesis of Daniel Kröning [Krö01].

3.2.2 The VAMP Design

In analogy to the specification we denote with c_I the implementation configuration of the VAMP. All components of the c_I we define later. The implementation of the VAMP has five separate execution units. There is one for the fixed point arithmetic *XPU*, three for the floating point arithmetic ($FPU1$ for addition/subtraction, $FPU2$ for multiplication/division and $FPU3$ for conversion/testing) and a memory unit *MU*. The design and correctness proof of three *FPU*s is omitted in this thesis and can be found in [Jac02a]. The fixed point unit is implemented as an *ALU* with a shifter. The correctness of this unit follows directly from the basic circuit correctness because there are no registers inside this unit. The implementation and correctness of the memory unit *MU* we will be described in Section 3.2.3.

Our implementation of the Tomasulo algorithm has eight reservation stations: four reservation stations for the fixed point unit and one reservation station for each other functional unit. We also have some instructions which use no functional unit, e.g. the instruction `movs2i` that copies data from a special purpose register to a general purpose register. We write these instructions in the reorder buffer directly after they are issued but, of course, since these instructions cannot snoop on the *CDB* they have to be stalled in the stage decode until all their operands are available.

The previous implementation of the VAMP had up to *six* 32-bit source operands. Six operands were only used for the floating point operations, four containing two 64-bit operands for double precision operands, the status register and the rounding mode register. For any memory access we have used at most three operands, one for the address and two 32-bit operands for 64-bit width data.

In our implementation of the VAMP we keep the upper limit of source operands but we extend the source operands in the case of any memory

access instruction with three new operands for page table origin, page table length and for mode bit. Therefore we have six source operands for any memory instruction. Note that since we keep the old limit of source operand these extensions do not enlarge the hardware of the VAMP. Alternatively, we can read *PTO*, *PTL*, and *MODE* registers directly from the *SPR* file. This approach, however, would require a more complicated proof.

In the previous VAMP the producer table for the special purpose register file was implemented as a *RAM* with 5-bit addresses and we could only change the valid bit and the tag for one different register at the same time. Since now the effect of `rfe?` command is $SPR[MODE] := SPR[EMODE]$ and $SPR[SR] := SPR[ESR]$, we need to update the producer table for two special-purpose registers at the same time. We use a similar approach as for the *FPR* register file and producer table. We split the producer table for special purpose register file into two *RAM*s with 4-bit addresses. In the first *RAM* we will have valid bits and tags for registers from 0 to 15 according to the table 3.1 and in the second *RAM* we will have only a valid bit and a tag for one register *MODE*, so we can change producer table for two *SPR* registers simultaneously as along as one of these registers is the *MODE* register. Note that we introduced the registers $S12$-$S15$ only in order to define all registers in the first *RAM*.

We also use the same fetch mechanism in the VAMP implementation as in the previous one. We have separate stages fetch and decode as shown in Figure 3.1. Note that we fetch instruction I_{i+1} and decode instruction I_i simultaneously in order to evaluate the branch condition prior to the next instruction fetch. Therefore we decided to use the delayed branch mechanism in the VAMP in analogy to the DLX from [MP00]. For this purpose in [MP00] was changed the semantics of the assembler instruction set such that all jumps and branches will be executed with a delay of one instruction, i.e. the next instruction after any jump or branch instruction will *always* be executed. More details and the correctness of instruction fetch we describe later.

After the fetch of an instruction we save the result in the $S1$ registers, which are the instruction register $c_I.S1.IR$, the exception flag for misalignment for instruction fetch $c_I.S1.imal$, and the flag for the page fault on instruction fetch $c_I.S1.ipf$. Later, in our correctness criteria we see that register $c_I.S1.IR$ is mapped to the function $IR(c_S)$ which is defined on the programmer's model, $c_I.S1.imal$ is mapped to $imal(c_S)$ and $c_I.S1.ipf$ to $CA(c_S)[ipf]$.

The previous VAMP had a mechanism for the internal interrupts. We extend this mechanism for external devices.

Following [Bey05] the exception cause $CA(c_I)$ and the exception data $EData(c_I)$ are parts of the result in the reorder buffer (only for internal interrupts) and we can compute $MCA(c_I)$, $JISR(c_I)$ and $repeat(c_I)$ during writeback in a way similar to the corresponding functions $MCA(c_S)$,

$JISR(c_S)$, and $repeat(c_S)$ in the programmer's model.

As long as we do not have any interrupts the VAMP works as the standard Tomasulo implementation, otherwise we make a so-called flush, i.e. we clear the instruction register, all reservation stations, functional units, producers, and the reorder buffer and mark all registers in the register file as valid. After that we start to execute the ISR according to Equations 3.19, 3.20, and 3.25 in the programmer's model.

An implementation configuration is a 17-tuple $c_I = (PC', DPC, M, GPR, FPR, SPR, S1, RS, P, ROB, ROBhead, ROBtail, ROBcount, MU, FPU1, FPU2, FPU3)$. The components which were not yet described are:

- $ROBhead$ – points to the head of the reorder buffer,
- $ROBtail$ – points to the tail of the reorder buffer,
- $ROBcount$ – contains the number of the entries, which are currently in the ROB. This counter is used in order to test whether the ROB is full or empty when $ROBhead = ROBtail$.

As before, we also do not have a component for the XPU because this functional unit contains no registers. Later we will use the same components in the implementation with an index I similar to the specification configuration. Following [Bey05], we define the memory content $c_I.M$ as follows:

$$
\begin{aligned}
c_I^0.M &:= init\_mem \\
\left|c_I^{t+1}.M[\langle ad\rangle]\right|_b &:= \begin{cases} \left|din^t\right|_b & \text{if } M_I.mbw(ad,b)^t \\ \left|c_I^t.M[\langle ad\rangle]\right|_b & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that we can apply decomposition to $c_I.M$, i.e. $c_I^{t+t'}.M = c_I[c_I^t]^{t'}.M$.

We also will use c_I^t for the configuration of hardware cycle t. We can make a definition for initial implementation configurations as in [Bey05].

Definition* 3.2.1 *We call an implementation configuration* c_I initial*, denoted* $init?(c_I)$*, iff all reservation stations, execution units, producer registers, the ROB, and the decode stage are empty and all registers are valid, in the program counter DPC the address SISR is saved and in* PC' *the address* $SISR + 4$ *is saved. Formally, we have:*

$$
\begin{aligned}
init?(c_I) \iff\ & \neg c_I.S1.full \wedge c_I.ROBcount = 0 \wedge \\
& empty?(c_I.MU) \wedge empty?(c_I.FPU1) \wedge \\
& empty?(c_I.FPU2) \wedge empty?(c_I.FPU3) \wedge \\
& c_I.DPC = SISR \wedge c_I.PC' = SISR + 4 \wedge \\
& (\forall x \in \mathbb{Z}_8 : \neg c_I.RS[x].full) \wedge (\forall x \in \mathbb{Z}_5 : \neg c_I.P[x].full) \wedge \\
& (\forall x \in \mathbb{Z}_{32} : c_I.GPR[x].valid \wedge c_I.FPR[x].valid) \wedge \\
& (\forall x \in \mathbb{Z}_{17} : c_I.SPR[x].valid)
\end{aligned}
$$

Note that bit $full$ for $c_I.RS[x].full$ is active iff this reservation station is full and bit $c_I.P[x].full$ is active iff this producer is full. Predicate empty? for any functional unit holds only if the current configuration of this functional unit is initial, i.e. configuration after power up or an interrupt. Note also that we do not need empty? for the XPU since the XPU is purely combinational. We will not give any details on the predicate empty? for FPUs since this would involve the complete pipeline structure of the three floating point units. The predicate empty? for MU we define in Section 3.2.3.

We define a function $spec\_conf$ that creates a specification configuration from an initial implementation configuration. This function just takes all the visible parts from the implementation, i.e., it is defined by the following equations:

$$\begin{aligned}
spec_conf(c_I).PC' &= c_I.PC' \\
spec_conf(c_I).DPC &= c_I.DPC \\
spec_conf(c_I).M &= c_I.M \\
spec_conf(c_I).GPR &= \lambda_{x\in\mathbb{Z}_{32}} c_I.GPR[x].data \\
spec_conf(c_I).FPR &= \lambda_{x\in\mathbb{Z}_{32}} c_I.FPR[x].data \\
spec_conf(c_I).SPR &= \lambda_{x\in\mathbb{Z}_{17}} c_I.SPR[x].data
\end{aligned}$$

3.2.3 Implementation of the VAMP Memory Unit

Figure 3.2 shows the extension of the previous VAMP with two *MMUs*, one for the instruction fetch (*IMMU*) and one for load / store (*DMMU*). The interface signals of the *MMUs* are named according to Definitions 1.5.1 and 1.5.3. For the two interfaces we use the same subscripts p and m as were introduced in Section 2.1 for CPU_I and M_I respectively.

The whole memory unit of the VAMP is presented in Figure 3.3 and consists of three parts:

1. The memory system M_I satisfying Definition 1.5.1 with address width $a = 29$ and 8 bytes per cell, $B = 8$.

2. *DMMU* which is a copy of the *MMU* from Chapter 2 and some auxiliary circuits for memory access processing. The auxiliary circuits (part of CPU_I) are:

   - circuit $shift4store$ is used to shift the data which have to be written to the correct byte position in a double word of the CPU_I interface. We use this circuit because the data input and output of memory is 64-bits wide and we need to access memory with $d \in \{1, 2, 4, 8\}$ bytes.
   - circuit *genbw* is used to generate the correct byte write signals *mbw* for the memory interface for store operations.

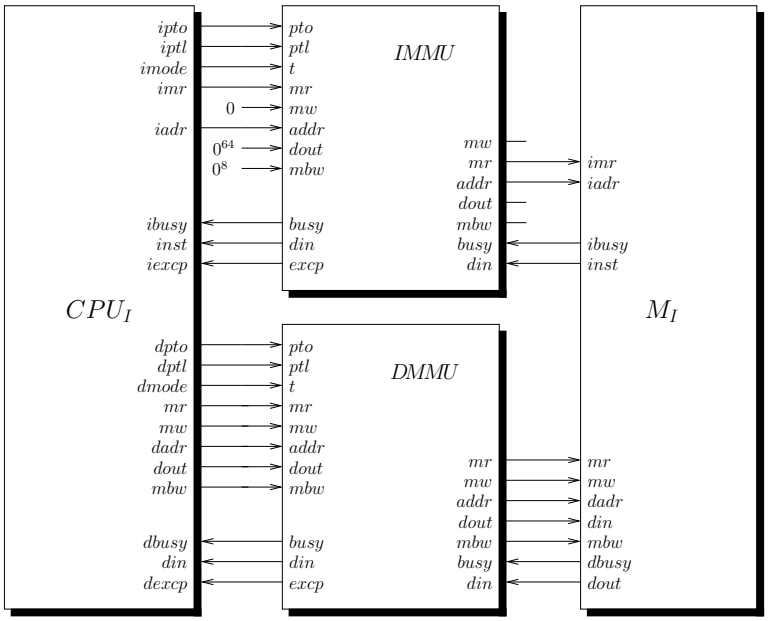

Figure 3.2: Extension of the VAMP with Two *MMU*s

- circuit $shift4load$ is used for the similar shifting as $shift4store$ but only for read accesses. Since both signed and unsigned loads are supported, the circuit $shift4load$ also performs sign-extension denoted as *sext* or zero-extension denoted as *zext* of the data which is loaded.
- circuit *comp_adr* is used to compute the effective byte address according to Definition 3.8.
- circuits $flags$, $flags2$, $flags3$ are used in order to compute control signals for load/ store memory access.

Complete definitions of these circuits can be found in [Bey05]. Implementations for these circuits (other than $flags$, $flags2$, $flags3$) are given in [MP00, p. 78–88].

3. *IMMU* which is a copy of the *MMU* from Chapter 2 and some auxiliary circuits which are part of CPU_I. We compute the signal *imr* and an instruction address *iadr* with the help of the circuit *gen_pc*. Since in the memory interface only 64 bits width operations are allowed and our instruction word is 32 bit wide we use a multiplexor in order to choose the correct instruction word. The construction of *gen_pc* we describe later.

Load / store instructions are processed as follows. On dispatch, the effective address *ea* is computed from two fields in the reservation station:

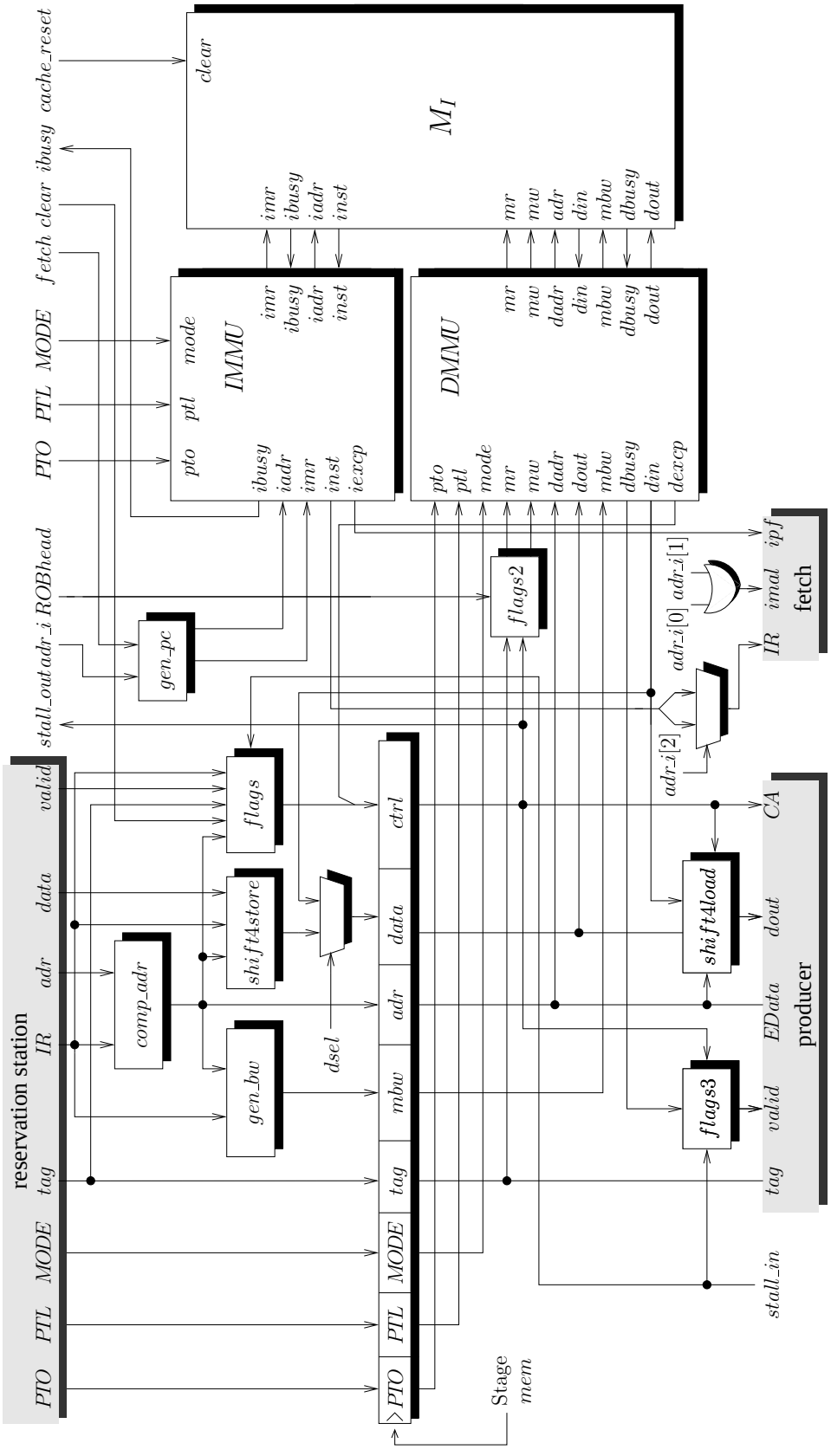

Figure 3.3: The VAMP Memory Unit

the instruction register which contains a copy of the instruction word and the $RS1$-operand in the reservation station. The address ea is written into the memory stage together with the shifted data from the circuit $shift4store$ for store instructions. The tag from the reservation station, the memory byte write signals from the circuit $genbw$ and some flags for the encoding of the type of the memory access are written into the memory stage as well. In case of misalignment ($dmal$=1) we must not access the memory at all. In the other case we can start a request to the *DMMU*. Note that in the case of store we also need to wait until the tag of the instruction will be equal to $ROBhead$. In this case we know that the store instruction is the oldest instruction in the pipeline what is used for organization of precise interrupts. The request to the *DMMU* takes at least three cycles. During this time the *DMMU* is busy and initiates up to two memory accesses but only the second access can be a write access. The memory stage is stalled and $stall\_out$ shows for the Tomasulo scheduler that the memory unit is not ready to accept the next instruction by dispatch.

When the *DMMU* finally signals the end of the request with $\neg dbusy_p$ we have two possibilities as in the previous VAMP. In the first case $stall\_in$ signals that the Tomasulo scheduler cannot currently accept outputs from the *DMMU* and we need to wait until the signal $stall\_in$ is inactive. Therefore in this case we store the result of the memory access in the intermediate registers: we store the exception flag $dexcp$ in the $mem.ctrl$ register and in the case of read access the data returned from the *DMMU* is stored into the register $mem.data$. As soon as $stall\_in$ is inactive the result of the instruction leaves the memory unit. For load instructions $data$ pass through the circuit $shift4load$ which shifts the double word data to the right and only returns the $d \in \{1, 2, 4, 8\}$ rightmost bytes of the data. Note that $shift4load$ returns either sign or zero extended shifted data.

Instruction fetch works as follows. In the case the input signal $fetch$ is active and we do not have any misalignment $imal$ we start the *IMMU* access with the address of instruction which is received from additional input adr_i (we define this input later). Page table origin, page table length, and mode are read directly from the *SPR* file. The request to the *IMMU* takes at least three cycles. During this time the *IMMU* can also initiate up to two memory accesses but all of them are load accesses. When the *IMMU* finally signals end of the access with $\neg ibusy_p$ we use $PC[2]$ in order to select the half of the double word data which has to be saved in the instruction register.

Note that we need to guarantee valid inputs according to the Definitions 1.5.1 and 1.5.3 for both *MMUs*. Therefore we add an additional stabilizing circuit in order to stabilize the *DMMU* inputs in case of an interrupt. With the help of this stabilizing circuit we can later show that predicate $p\_req\_is\_stable$ holds form Section 2.1.1. We introduce an additional control bit $rollback$ to the memory stage which is only active in the case when an interrupt $JISR(c_I)$ has occurred for during a request to the *DMMU*. The

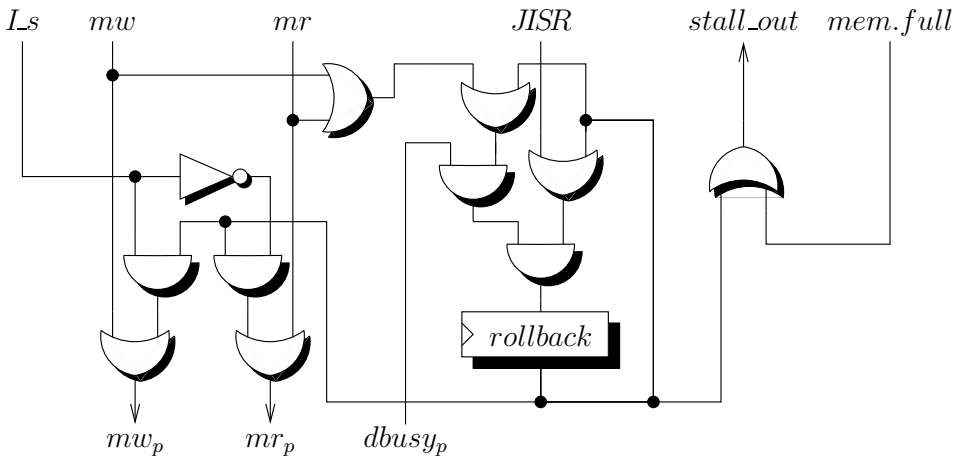

Figure 3.4: Stabilizing Circuit for the *Data MMU*

control bit *rollback* stays active until the end of an interrupted request, i.e. until the *DMMU* signals $\neg dbusy_p$. Note that in this case we do not use the result of an interrupted request. The memory unit has to activate the signal $stall\_out$ as long as *rollback* is active. As long as the bit *rollback* is active one of the signals mr_p or mw_p is also active. With help of the signal I_s we encode the type of the access: the signal I_s is active for store operations and we prove later that store accesses cannot be interrupted. The stabilizing circuit is defined in Figure 3.4.

The following equations describe the behavior of stabilizing circuit according to Figure 3.4:

$$\begin{aligned} c'_I.rollback &= (mr \vee mw \vee rollback) \wedge (JISR(c_I) \vee rollback) \wedge dbusy \\ stall\_out &= mem.full \vee rollback \\ mr_p &= mr \vee rollback \wedge \neg mem.I\_s \\ mw_p &= mw \vee rollback \wedge mem.I\_s \end{aligned} \tag{3.25}$$

where

- mw is a signal from the circuit $flags2$ and is active when the memory stage is full, we do not have misalignment and we decode the instruction in the memory stage as a store instruction.
- mr is a signal from the circuit $flags2$ and is active when the memory stage is full, we do not have misalignment and we decode the instruction in the memory stage as a load instruction. Note that in case when $mw \vee mw$ holds we have $\neg dmal(c_I)$ and signal $full$ is active.
- $full$ is a signal which indicates that in the memory unit there currently is an instruction. Note that for all cycles $t \in \mathbb{N}$ we know that

1. $mr^t \vee mw^t \implies mem.full^t$,
2. $\neg(mem.full^t \wedge c_I^t.rollback)$.

Both properties follows directly from the construction of the *MU* and were proved before in [Bey05].

For the stabilizing circuit *genPC* based on the same principle, we add an additional control bit *irollback* in order to stabilize the read signal imr_p. We also have to add an additional register mPC that stores the PC in the cycle when the signal $JISR(c_I)$ is active during an access to the *IMMU* (see Figure 3.5). In the case that an interrupt occurs during the request to the *IMMU* the control bit *irollback* is raised and the *IMMU* starts to read the address of request from the mPC. The control bit *irollback* gets inactive in the next cycle after the end of an access to the *IMMU*, i.e. after $\neg ibusy_p$. The following equations summarize the behavior of the signals from Figure 3.5.

$$\begin{aligned}
irollback' &= (JISR(c_I) \wedge (\neg imal \wedge fetch \vee irollback) \\
&\quad \vee \neg JISR(c_I) \wedge irollback) \wedge ibusy_p \\
mPC' &= \begin{cases} PC[31:3] & \text{if } ibusy_p \wedge \neg irollback \\ mPC & \text{otherwise} \end{cases} \\
imr_p &= \neg imal \wedge fetch \vee irollback \\
iadr_p &= \begin{cases} mPC & \text{if } irollback \\ PC[31:3] & \text{otherwise} \end{cases}
\end{aligned} \tag{3.26}$$

Now we give the construction of the address computation for instruction fetch which was not changed from [Bey05]. We define the $adr\_i(c_I)$ which returns the address used in the instruction fetch implementation according to Figure 3.6.

$$adr\_i(c_I) := \begin{cases} c_I.SPR[EDPC] & \text{if } c_I.S1.full \wedge \texttt{rfe}?(c_I.S1.IR) \\ c_I.PC' & \text{if } c_I.S1.full \wedge \neg\texttt{rfe}?(c_I.S1.IR) \\ c_I.DPC & \text{otherwise} \end{cases} \tag{3.27}$$

We introduce the predicate *empty*? for the memory unit. This predicate holds if

- no instruction is executed in the memory unit,
- initial state of control automaton of the *DMMU* is unique, i.e. the control cannot be in two different states at the same cycle. For this purpose we define predicate is_unary? which holds in this case,

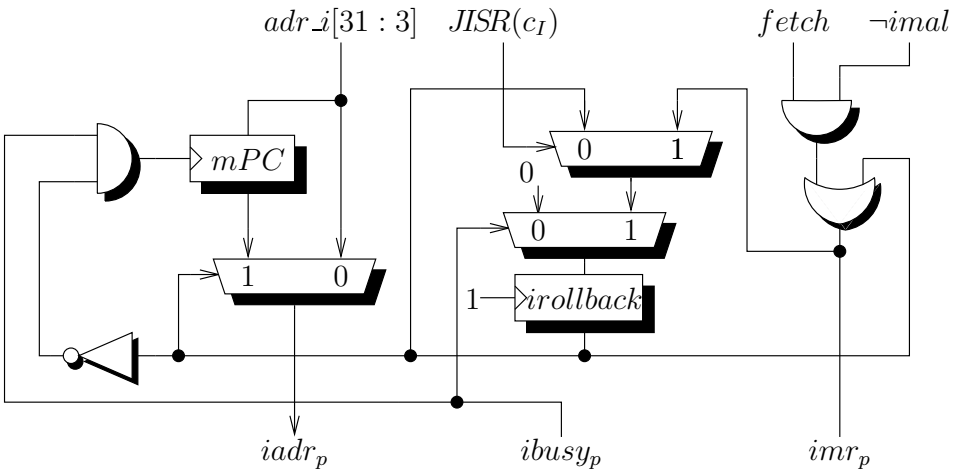

Figure 3.5: Stabilizing Circuit for the *Instruction MMU* (Circuit *genPC*)

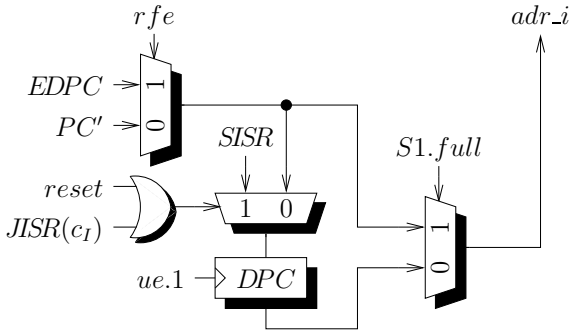

Figure 3.6: Fetch Implementation in the VAMP.

- initial state of control automaton of the *IMMU* is unique, i.e. the control cannot be in two different states at the same cycle,
- in the memory unit is not executed an interrupted read instruction,
- in the memory unit is not executed an interrupted fetch, and
- write request can not be interrupted.

Formally we have:

$$\begin{aligned} empty?(c_I.MU) := \neg full \\ \wedge is\_unary?(state_{DMMU}) \\ \wedge is\_unary?(state_{IMMU}) \\ \wedge(\neg rollback \implies state_{DMMU}.idle) \\ \wedge(\neg irollback \implies state_{IMMU}.idle) \\ \wedge(rollback \implies \neg I\_s) \end{aligned}$$

3.3 Correctness Criteria

There are two basic concepts to organize correctness criteria for out-of-order processors.

1. The first is based on the so-called "flushing" technique. At any time there is some number of instructions in the processor pipeline. We can execute these instructions while no new instructions are initiated, i.e. while we stall the processor fetch. Then after a certain number of cycles we have executed all instructions which were in the pipeline; this is called a "flushing" operation. The basic idea of this correctness criteria is as follows. At start time the implementation of the processor has a configuration c_I^j. Then we flush the pipeline and after that we can get for some instruction i the specification configuration c_S^i after executing all the instructions by applying $spec\_conf$ function. We can do it because if the pipeline is empty and implementation of the processor is correct the specification configuration is one part of the implementation configuration. We can now execute one more instruction $i+1$ on the specification configuration and we get the next specification configuration c_S^{i+1}. We also have another possibility to calculate c_S^{i+1}: we do not stall the pipeline until the next instruction $i+1$ begin to execute. We now flush the pipeline and after flushing we can calculate the new specification configuration $\hat{c}_S^{i+1}$. The implementation of our processor satisfies the specification iff $c_S^{i+1} = \hat{c}_S^{i+1}$. This technique was described in [BD94] and is widely used for automatic hardware verification. This correctness criterion has been applied ito many designs, e.g., by Hosabettu, et al. [HSG99], and Velev and Bryant [VB00].

Name	Scheduling function
$fetch$	Scheduling function for the stage fetch.
$decode$	Scheduling function for the stage decode.
$issue$	Scheduling function for the issue.
R	Scheduling function for every visible register R except the memory.
mem	Scheduling function for the stage mem which is inside the memory unit MU.
M_I	Scheduling function for the visible memory.
wb	Scheduling function for the stage writeback.
$inst$	Scheduling function for the visible register files for correctness with interrupt.

Table 3.3: Scheduling Functions of the VAMP

2. The second concept is based on scheduling functions. Scheduling functions map a pipeline stage in some cycle to the index of the instruction according to the specification. They were introduced in [MP00, BJK+03]. A similar idea was used by Sawada and Hunt in [SH98,HS99], which is based on Micro-Architectural Execution Trace Table.

   Definition 3.3.1 *If k is some stage in the CPU and t is a cycle, a scheduling function $sI(k,t)$ returns a natural number which we associate with the index of the instruction that is in stage k in cycle t.*

   We decided to use this approach because the previous proof of VAMP was based on scheduling functions.

3.3.1 Scheduling Functions

Since we do not change the scheduling functions we only give definitions of scheduling functions. We have only changed signals which we use for the computation of scheduling functions. More detail about scheduling functions can be found in [Bey05, p. 126–132].

At first we give the definitions for scheduling functions without interrupts. In Table 3.3 we show some scheduling functions which are used in this thesis. Scheduling functions are defined inductively over cycle t. If an instruction at time t in stage k is passed to the stage k' we have $sI(k', t+1) = sI(k,t)$. We have $sI(k,0) = -1$ in order to model an empty pipeline for the initial cycle.

We introduce the scheduling functions according to Table 3.3. We start with the scheduling function for the stage *decode*. Scheduling functions for

stages fetch, decode and issued are introduced in [Bey05] with the help of the following signals:

- $stall.1$ a signal which indicates that in the next cycle we cannot accept inputs for the stage *decode*.
- $ue.0$ and $ue.1$ update enable signals for instruction fetch and decode correspondingly. Signal $ue.0$ indicates that an instruction fetch is completed and signal $ue.1$ indicates that the instruction in the instruction register is being issued in the next cycle into one of the reservation stations or directly into the reorder buffer.
- signal $S1.full$ is active if the stage *decode* cannot accept inputs from the stage fetch. It is defined in the following way:

$$\begin{aligned} S1.full^0 &= 0 \\ S1.full^{t+1} &= ue.0^t \vee stall.1^t \end{aligned}$$

  where $t \in \mathbb{N}$.

For fetching the next instruction we need to use additionally registers from the special purpose register file. They are $SPR[PTO]$, $SPR[PTL]$ and $SPR[MODE]$. These values are taken directly from the register file. Therefore we have to know that no instructions are being executed in the pipeline which change one of these registers. For this purpose we introduce a new signal $fetch$. This signal is inactive when

- in the pipeline there is an instruction writing to one of these registers
- the instruction in the stage *decode* is `rfe` or `movi2s`, one of the instructions which may change the registers *PTO*, *PTL*, or *MODE*.

Additionally, the `sync` instruction stalls the fetch until all previous instructions have left the pipeline. More detail about the `sync` instruction we give in Section 3.4.2. Formally we define:

$$\begin{aligned} fetch \;:=\; & SPR[PTO].valid \wedge SPR[PTL].valid \wedge SPR[MODE].valid \wedge \\ & \neg(S1.full \wedge (\texttt{rfe}?(S1.IR) \vee \texttt{movi2s}?(S1.IR) \vee \texttt{sync}?(S1.IR)) \end{aligned}$$

Using the $fetch$ signal, we change the old definition of the $ue.0$ signal to

$$ue.0 \;=\; \neg ibusy_p \wedge \neg stall.1 \wedge fetch \wedge \neg irollback$$

Using this signal the definition of the scheduling function for the stage decode is as follows:

$$\begin{aligned} sI(dec, 0) &:= -1 \\ sI(dec, t+1) &:= \begin{cases} sI(dec,t)+1 & \text{if } ue.0^t \\ sI(dec,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.28}$$

For the issue scheduling function:

$$\begin{aligned} sI(issue,0) &:= 0 \\ sI(issue,t+1) &:= \begin{cases} sI(issue,t)+1 & \text{if } ue.1^t \\ sI(issue,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.29}$$

The scheduling function for the stage fetch is defined as $sI(fetch,t) = sI(dec,t)+1$. The scheduling function for every visible register R except the memory is as follows:

$$\begin{aligned} sI(R,0) &:= 0 \\ sI(R,t+1) &:= \begin{cases} sI(k,t)+1 & \text{if } ue.k^t \\ sI(R,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.30}$$

We have two scheduling functions for the memory. The first scheduling function is $sI(mem,t)$ for the stage mem, which is inside the memory unit MU. The definition of this function is straightforward: If the MU accepts a new instruction in cycle t from the reservation station the scheduling function $sI(mem,t+1)$ is set to the index of that instruction. Otherwise the scheduling function remains unchanged. Note that we initialize this scheduling function with -1, i.e. $sI(mem,0) = -1$.

The second scheduling function is the scheduling function for the visible memory which is defined as follows:

$$\begin{aligned} sI(M_I,0) &:= 0 \\ sI(M_I,t+1) &:= \begin{cases} sI(mem,t)+1 & \text{if } (mw^t \vee mr^t) \wedge \neg dbusy_p^t \\ sI(M_I,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.31}$$

The scheduling function for the stage writeback is as follows:

$$\begin{aligned} sI(wb,0) &:= 0 \\ sI(wb,t+1) &:= \begin{cases} sI(wb,t)+1 & \text{if } ue.wb(c_I^t) \\ sI(wb,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.32}$$

where the $ue.wb(c_I^t)$ signal is update enable signal for the writeback stage. In the end we need some scheduling function counting completed instructions and interrupts in order to claim correctness with interrupts. This scheduling function is defined as:

$$\begin{aligned} sI(inst,0) &= 0 \\ sI(inst,t+1) &= \begin{cases} sI(inst,t)+1 & \text{if } ue.wb(c_I^t) \vee JISR(c_I^t) \\ sI(inst,t) & \text{otherwise} \end{cases} \end{aligned} \tag{3.33}$$

Of course, the following proposition trivially holds.

Proposition* 3.3.2 *Until one cycle* after *the first interrupt, the scheduling function counting instructions and interrupts and the writeback scheduling function are equal, i.e., for all* $t \in \mathbb{N}$*:*

$$(\forall k \in \mathbb{Z}_t : \neg JISR(c_I^k)) \implies sI(inst, t) = sI(wb, t)$$

3.3.2 Correctness Invariant

We define correctness criteria for computation without interrupts and for computation with interrupts. The former will be used for the proof of the correctness of our VAMP between interrupts. The latter will be used for the proof of the correctness of our VAMP with interrupts.

Definition 3.3.3 *We call a VAMP implementation configuration* correct without interrupts *in cycle* t*, denoted,* $corr?(t)$*, iff under the assumption* $\forall t' \in \mathbb{Z}_{\leq t} : \neg JISR(c_I^{t'})$ *the following conditions hold*

1. $c_I^t.M = \tilde{c}_S^{sI(M_I,t)}.M$
2. $c_I^t.PC' = \tilde{c}_S^{sI(issue,t)}.PC'$
3. $c_I^t.DPC = \tilde{c}_S^{sI(issue,t)}.DPC$
4. $sI(dec, t) \geq 0 \implies c_I^t.S1.IR = IR_S(\tilde{c}_S^{sI(dec,t)}) \wedge$
   $c_I^t.S1.imal = imal(\tilde{c}_S^{sI(dec,t)}) \wedge$
   $c_I^t.S1.ipf = ipf(\tilde{c}_S^{sI(dec,t)})$
5. $\forall x \in \mathbb{Z}_{32} : c_I^t.GPR[x].data = \tilde{c}_S^{sI(wb,t)}.GPR[x]$
6. $\forall x \in \mathbb{Z}_{32} : c_I^t.FPR[x].data = \tilde{c}_S^{sI(wb,t)}.FPR[x]$
7. $\forall x \in \mathbb{Z}_{17} : c_I^t.SPR[x].data = \tilde{c}_S^{sI(wb,t)}.SPR[x]$

Correctness criteria for computation with interrupts is following:

Definition 3.3.4 *We call a VAMP implementation configuration* correct with interrupts *in cycle* t*, i.e.,* $corr\_i?(t)$*, iff the following conditions hold:*

1. $(t = 0 \vee JISR(c_I^{t-1})) \implies c_I^t.M = c_S^{sI(inst,t)}.M$
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. $\forall x \in \mathbb{Z}_{17} : c_I^t.SPR[x].data = c_S^{sI(inst,t)}.SPR[x]$

Note that $corr\_i?$ and $spec\_conf$, which was introduced in Section 3.2.2, are related in the following way:

$$(t = 0 \vee JISR(c_I^{t-1})) \wedge corr\_i?(t) \implies spec\_conf(c_I^t) = c_S^{sI(inst,t)}$$

3.3.3 Proof Overview

The structure of the proof is the same as it was in the VAMP without address translation. In this thesis we present the parts of the whole proof which are changed or added to the previous proof. Following [Bey05] we start with the correctness proof of the VAMP without interrupts, then we extend it to deal with interrupts.

The whole proof which we present in this thesis consists of three parts:

1. We prove the correctness of the memory access with address translation in the case when we have load or store instructions (see Figure 3.2).
2. In the second part we describe the proof for the fetch mechanism which was extended with the *Instruction MMU* (*IMMU*). Note that the *data MMU* and the *instruction MMU* are identical but we use the *IMMU* for read accesses only (see Figure 3.2).
3. Later we show the correctness of the mechanism dealing with external interrupts which allows to connect I/O devices to the VAMP.

Definition 3.3.5 *We use short notation $dinputs_x$ ($iinputs_x$) for all signals which are inputs to the DMMU (IMMU) and $doutputs_x$ ($ioutputs_x$) for all signals which are outputs from the DMMU (IMMU) where x denotes the interface $x = p$ for CPU_I and $x = m$ for M_I.*

3.4 Correctness between Interrupts

In this section we introduce the correctness proof of the VAMP implementation without interrupts. We will show the induction step for the VAMP correctness without interrupts only for the parts which were changed, i.e. for the memory unit and for the instruction fetch which was extended with two *MMU*s.

We introduce the basic lemma for the correctness without interrupts. In the following lemma we use the predicate $synced\_code?$ in order to deal with self-modifying code. It is needed since an instruction which has been now fetched at a certain time could be overwritten by a store instruction still in the pipeline. Note that we will give the formal definition of the predicate $synced\_code?$ in Section 3.4.2 where we consider instruction fetch.

Lemma 3.4.1 *The VAMP implementation is correct without interrupts. Formally, we have for all* $t \in \mathbb{N}$*:*

$$(\forall t' \in \mathbb{Z}_{\leq t} : \neg JISR(c_I^{t'})) \wedge synced\_code? \implies corr?(t)$$

In order to prove this lemma we show the validity of all parts of the predicate *corr*? which were changed.

3.4.1 Correctness of the Memory Unit on Load / Store

We show the correctness of the memory unit as follows:

- First, the memory unit in cycle t produces the same outputs as the corresponding part of the specification, i.e. the correct data is read in the case of a load operation. We focus on the cycle when a request actually completes. Note that we do not change the other part of the proof when we have to store the result from the *DMMU* in the memory unit and pass it to the producer later on an inactive *stall_out*.

- Second, the memory in the next cycle $t + 1$ fulfills the correctness invariant, i.e. the correct data is stored in the memory according to the correct address in the case of a store instruction. We also focus only on the cycle when a request actually completes.

Note that for both cases we will assume that the predicate $corr?(t')$ holds for all cycles $t' \leq t$ and we do not have interrupts up to time t'. The correctness criteria of the Tomasulo algorithm guarantee the correctness of the information in the memory reservation station. With help of this information we can easily conclude the correctness of the registers in the memory stage, i.e. the effective address, the data to be stored, and several flags.

In both cases we focus on aligned accesses since correctness of misaligned accesses is trivial to show.

In the correctness proof of the memory unit we want to use the correctness of the local *MMU*, which was shown in Chapter 2. For this purpose we have to define five components that constitute a local *MMU* computation. These are the memory sequence, inputs and outputs sequences for the processor, and inputs and outputs sequences for the memory system. As in Chapter 2 these five components induce a trace *dtrc* for the *DMMU*.

The idea behind these sequences is as follows. From the VAMP computation we only take the request to the *DMMU* we are interested in. The local *MMU* computation is constituted such that this request is placed at the beginning of the local *MMU* computation and no more requests follow afterwards. Figure 3.7 shows the construction of the memory sequence for the *DMMU* from the memory sequence of the VAMP and construction of the request signal. The start cycle t' of the processor request to the *DMMU*

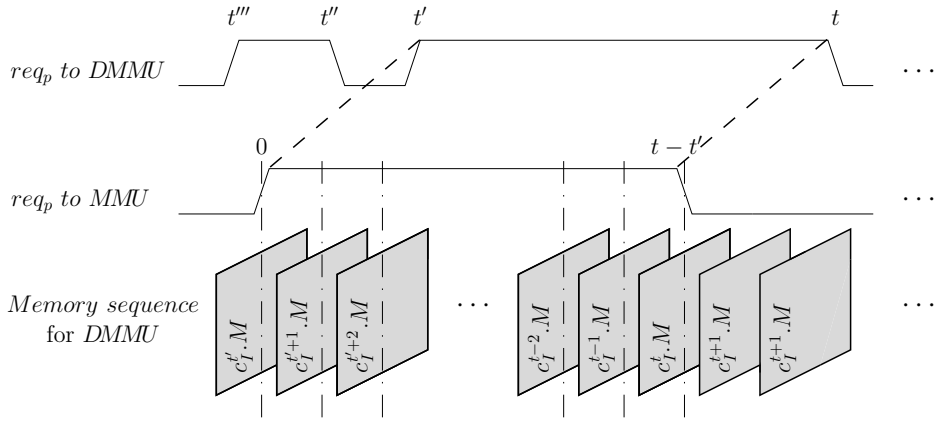

Figure 3.7: Memory sequence for the *DMMU*

represents the local *MMU* cycle 0. The last cycle t of the processor request to the *DMMU* represents the local *MMU* cycle $t - t'$.

Formally, we define:

- Memory sequence

$$dtrc.mem = \lambda_{t'' \in \mathbb{N}}. \begin{cases} c_I^{t''+t'}.M & \text{if } t' + t'' \leq t \\ c_I^{t''+t'+1}.M & \text{otherwise} \end{cases}$$

  Note that for local correctness of the *MMU* we need one port memory but in our case we have two ports memory. We can have only read accesses on instruction port according to the Definition 1.5.1. Therefore data port interface of two ports memory presents the same interface as for one port memory.

- Inputs sequence for inputs from the processor

$$dtrc.in_p = \lambda_{t'' \in \mathbb{N}}. \begin{cases} dinputs_p^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$

  where X is an arbitrary $dinputs_p$ such that mr_p and mw_p are always inactive.

- Inputs sequence for inputs from the memory system

$$dtrc.in_m = \lambda_{t'' \in \mathbb{N}}. \begin{cases} dinputs_m^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$

  where X is an arbitrary $dinputs_m$ such that $dbusy_m$ is always inactive.

- Outputs sequence for outputs to the processor

$$dtrc.out_p = \lambda_{t'' \in \mathbb{N}}. \begin{cases} doutputs_p^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$

  where X is an arbitrary $doutputs_p$ such that $dbusy_p$ and $dexcp_p$ are always inactive.

- Outputs sequence for outputs to the memory system

$$dtrc.out_m = \lambda_{t'' \in \mathbb{N}}. \begin{cases} dinputs_m^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$

  where X is an arbitrary $dinputs_m$ such that mr_m and mw_m are always inactive and $daddr_m = daddr_m^{t''+1}$.

Note that later, in order to use correctness of the *MMU* from Chapter 2, it will be not enough only to prove all the assumptions. We also have to show that this computation is a computation of the local *MMU*. It is easy to show that these definitions constitute a valid local *MMU* computation.

We prove that local *MMU* correctness which were introduced in Section 2.1 holds for the *DMMU*.

Lemma 3.4.2 *:*

$$\forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', dtrc) \wedge corr?(t') \implies mmu\_guarantee(dtrc)$$

In order to prove this lemma we need one additional lemma.

Lemma 3.4.3 *All inputs to the DMMU are stable if the DMMU is busy and $mr_p^t \vee mw_p^t$ is active in the same cycle:*

$$\forall t \in \mathbb{N} : (mr_p^t \vee mw_p^t) \wedge dbusy_p^t) \implies dinputs_p^t = dinputs_p^{t+1}$$

Proof: Inputs $dinputs_p$ are taken directly from the stage mem. The proof follows directly from the construction of the memory unit since information in the stage mem could be changed only if $dbusy_p$ is inactive. □

Proof: [Lemma 3.4.2] The predicate $mmu\_guarantee(dtrc)$ consists of seven predicates. Based on Lemma 3.4.3 we conclude that the predicate $p\_req\_is\_stable(dtrc)$ holds until the end of request to the *DMMU*. After the request we always have $\neg mw_p$ and $\neg mr_p$, thus $p\_req\_is\_stable(dtrc)$ holds for the whole trace.

The second assumption of the local correctness for the processor interface is that mr_p and mw_p are mutually exclusive, i.e. $p\_mr\_mw\_mutexc(dtrc)$.

This property is obvious to prove since mr_p is active only if $\neg I\_s$ holds and mw_p is active only when I_s holds (Figure 3.4, Equation 3.25). For all cycles after the request to the *DMMU* the predicate $p\_mr\_mw\_mutexc(dtrc)$ also holds since mr_p and mw_p are both inactive. Since we know that predicates $p\_req\_is\_stable(dtrc)$ and $p\_mr\_mw\_mutexc(dtrc)$ hold we conclude that the predicate $good\_p\_interface(dtrc)$ also holds.

From Definition 1.5.2 correctness of the predicates $m\_ack\_write(dtrc)$, $m\_read\_consist(dtrc)$, $m\_write\_consist(dtrc)$, and $m\_liveness(dtrc)$ during the request to the *DMMU* directly follows. Since mr_p and mw_p are both inactive after the request to the *DMMU* the control of the *DMMU* is in state *idle*. Then we conclude that mw_m and mr_m are also inactive. Since mw_m and mr_m are inactive all these predicates trivially hold. Therefore we know that the predicate $good\_m\_interface(dtrc)$ also holds.

So we have already proven all the assumptions for the *DMMU*. Therefore we know the correctness of the *DMMU* according to Lemma 2.3.1, i.e. $mmu\_guarantee(dtrc)$. □

We define the predicate which holds only in cycles when a request of the *DMMU* actually completes.

Definition 3.4.4 *We define the predicate commits for any $t, i \in \mathbb{N}$ in the following way:*

$$commits(t,i) := \\ (mw_p^t \vee mr_p^t) \wedge \neg dbusy_p^t \wedge sI(mem,t) = i$$

We reconstruct from the last cycle of an *DMMU* request the whole request.

Lemma 3.4.5 *If an instruction in the memory unit commits then we have a cycle t' before the cycle t such that from t' to t we have a request to the DMMU. Formally, we have:*

$$\forall t, i \in \mathbb{N} : commits(t,i) \implies \\ \exists t' \in \mathbb{Z}_{\leq t} : is\_req\_proc(t', t, dtrc)$$

In order to prove this lemma we have to prove one additional lemma.

Lemma 3.4.6 *When at any cycle t the DMMU is busy and this request was not interrupted (as indicated by the rollback signal) then we have a cycle t' before cycle t when the DMMU is not busy and from t' to t the memory unit is full and the DMMU is busy.*

$$\forall t \in \mathbb{N} : dbusy_p^t \wedge \neg rollback^t \implies \\ \exists t' \in \mathbb{Z}_{\leq t} : \neg dbusy_p^{t'} \wedge \\ \forall t'' \in]t' : t] : (dbusy_p^{t''} \wedge full^{t''})$$

Proof: We show the claim by induction on t':
Induction base ($\mathbf{t = 0}$): The predicate $init?(c_I)$ holds always in cycle 0. Therefore the predicate $empty?(c_I.MU)$ also holds. Since we have $\neg c_I^0.rollback$ the control automaton of the *DMMU* is in state *idle* by definition of $empty?(c_I.MU)$. We also have $\neg mem.full^0$ by definition of $empty?(c_I.MU)$. Since the control is in state *idle* and we have $\neg(mr_p^0 \vee mw_p^0)$ we conclude that $\neg dbusy_p^0$ and so we finish the base case.
Induction step ($\mathbf{t \rightarrow t+1}$): We split cases on $dbusy_p^t \wedge \neg rollback^t$:

1. Let $dbusy_p^t \wedge \neg rollback^t$ hold. From the induction hypothesis we know that:

$$\begin{aligned}&\exists t' \in \mathbb{Z}_{\leq t} : \neg dbusy_p^{t'} \wedge \\ &\quad \forall t''' \in]t' : t] : (dbusy_p^{t'''} \wedge mem.full^{t'''})\end{aligned}$$

   Additionally we can assume that:

$$dbusy_p^{t+1} \wedge \neg c_I^{t+1}.rollback$$

   We have to prove that:

$$\begin{aligned}&\exists t' \in \mathbb{Z}_{\leq t+1} : \neg dbusy_p^{t'} \wedge \\ &\forall t'' \in]t' : t+1] : dbusy_p^{t''} \wedge mem.full^{t''}\end{aligned}$$

   We skolemize the existential quantifier in the induction hypothesis. After the skolemization we have a cycle t'. After that we instantiate the last existential quantifier with cycle t'. In order to prove this case we only have to show that $mem.full^{t+1}$ is holds. Since we have $\neg dbusy^{t'} \wedge dbusy_p^{t'+1}$ we know that $mr_p^{t'+1} \vee mw_p^{t'+1}$. Since processor inputs are stable between cycles $t'+1$ and $t+1$ (Lemma 2.1.8) we have $mr_p^{t+1} \vee mw_p^{t+1}$. This concludes the claim since we have $\neg c_I^{t+1}.rollback$. Note that definition of $c_I'.rollback$ was introduced in Equation 3.25.

2. Let $\neg(dbusy_p^t \wedge \neg rollback^t)$ hold. We split cases on $dbusy_p^t$:

   (a) If $dbusy_p^t$ holds we have $rollback^t \wedge \neg rollback^{t+1}$. This is a contradiction because from the construction of $rollback$ we have $dbusy_p^t \wedge rollback^t \implies rollback^{t+1}$. This concludes this claim.

   (b) Let $\neg dbusy_p^t$ hold. We only have to prove that $full^{t+1}$ holds. Since we have $\neg dbusy_p^t$ and $dbusy_p^{t+1}$ we conclude $mr_p^{t+1} \vee mw_p^{t+1}$. This concludes the proof since we have $\neg rollback^{t+1}$. □

We can now prove Lemma 3.4.5

Proof: [Lemma 3.4.5] We show the claim by induction on t:

$$\forall t, i \in \mathbb{N} : commits(t, i) \implies \exists t' \in \mathbb{Z}_t : is\_req\_proc(t', t, dtrc)$$

Induction base ($\mathbf{t = 0}$): Since $sI(mem, 0)$ equal -1 we have $i = -1$. This is a contradiction because $i \geq 0$ and so we finish the base case.
Induction step ($\mathbf{t \to t + 1}$): We now split cases on $commits(t, i)$:

1. Let $commits(t, i)$ hold. We know that

   $$commits(t, i) \wedge commits(t + 1, i) \wedge \exists t' \in \mathbb{Z}_{\leq t} : is\_req\_proc(t', t, dtrc)$$

   We have to prove that

   $$\exists t' \in \mathbb{Z}_{\leq t+1} : is\_req\_proc(t', t + 1, dtrc).$$

   We instantiate the last existential quantifier with $t + 1$. All parts of the predicate $is\_req\_proc(t + 1, t + 1, dtrc)$ may then be concluded using the assumptions $commits(t, i)$ and $commits(t + 1, i)$. We finish this case.

2. Let $\neg commits(t, i)$ hold. In this case we know

   $$\neg commits(t, i) \wedge commits(t + 1, i)$$

   and we have to prove that:

   $$\exists t' \in \mathbb{Z}_{\leq t+1} : is\_req\_proc(t', t + 1, dtrc)$$

   We now split cases on $busy_p^t$:

   (a) Let $\neg busy_p^t$ hold. After instantiation of the existential quantifier with cycle $t + 1$ we split cases on all parts of predicate

   (b) Let $busy_p^t$ hold. In this case we have to prove that

   $$\neg commits(t, i) \wedge commits(t + 1, i) \wedge dbusy_p^t \implies \exists t' \in \mathbb{Z}_{\leq t+1} : is\_req\_proc(t', t + 1, dtrc)$$

   We split cases on $rollback^{t+1}$:

   i. If $rollback^{t+1}$ holds then $sI(mem, t + 1) = -1$ what is contradiction because we have $sI(mem, t + 1) = i$ and $i \geq 0$. It finishes this case.

ii. Let $\neg rollback^{t+1}$ hold. Since $dbusy_p^t$ is active we conclude that $rollback^{t+1} = rollback^t$ and so we have $\neg rollback^t$. Hence, the assumptions of Lemma 3.4.6 hold. By this lemma we know that there exists a cycle $tt < t$ such that $dbusy_p^{tt}$ is inactive and all cycles between tt and t signals $dbusy_p$ and $full$ are active. Of course, cycle $tt+1$ is the cycle when the request to the *DMMU* starts, i.e., we instantiate the existential quantifier with cycle $tt+1$ and so we finish the proof. □

We repeat a helper lemma which was introduced in [Bey05]. It states that the specification memory that the instruction in the memory stage sees at the end of an memory access is actually the specification memory of instruction $sI(M_I, t)$.

Lemma* 3.4.7 *If an instruction in the memory unit terminates its access in cycle t, the specification memory for the memory unit and the one of the instruction identified by the visible memory register scheduling function are identical. Formally, we have*

$$(mw^t \vee mr^t) \wedge \neg dbusy_p^t \implies \tilde{c}_S^{sI(M_I,t)}.M = \tilde{c}_S^{sI(mem,t)}.M$$

Proof of this lemma can be found in [Bey05, p. 148]. We now show that the memory unit in cycle t produces the same outputs as the corresponding part of the specification, i.e. the correct data is read in the case of a load operation.

Lemma 3.4.8 *Let the CPU correctness invariant without interrupts (Definition 3.3.3) hold in cycle t, let a load instruction $i = sI(mem, t)$ in the memory unit complete its aligned access in cycle t on the address $ea(c_S)$ with access width $d \in \{1, 2, 4, 8\}$, and let $r\_sext = \texttt{lb}?(IR(\tilde{c}_S^i)) \vee \texttt{lh}?(IR(\tilde{c}_S^i))$, and finally assume $\neg dexcp_p^t$. This load instruction then delivers the correct result, i.e., we have $corr?(t) \wedge mr_p^t \wedge \neg dbusy_p^t \wedge \neg dexcp_p^t \implies$*

$$MU.dout^t = \begin{cases} sext(\tilde{c}_S^i.M[ea(c_S)+d-1 : ea(c_S)]) & \text{if } r\_sext \wedge \neg t \\ zext(\tilde{c}_S^i.M[ea(c_S)+d-1 : ea(c_S)]) & \text{if } \neg r\_sext \wedge \neg t \\ sext(\tilde{c}_S^i.M[r\_t(t).pa+d-1 : r\_t(t).pa]) & \text{if } r\_sext \wedge t \\ zext(\tilde{c}_S^i.M[r\_t(t).pa+d-1 : r\_t(t).pa]) & \text{otherwise} \end{cases}$$

Proof: Let an aligned load instruction in the memory unit finish its access to the *DMMU* in cycle t without an exception, i.e., $mr_p^t \wedge \neg dbusy_p^t \wedge \neg dexcp_p^t$ holds. With the help of Lemma 3.4.7 we know that $\tilde{c}_S^{sI(mem,t)}.M = \tilde{c}_S^{sI(M_I,t)}.M$. Since predicate $corr?(t)$ holds we have $\tilde{c}_S^i.M = c_I^t.M$. Moreover, it is easy to conclude the correctness of the data in the memory stage, i.e. $ea(\tilde{c}_S^i) = ea(c_I^t)$. With the help of Lemma 3.4.5 we know that there

exists a cycle t' such that $t' \leq t$ and the predicate $is\_reg\_proc(t', t, dtrc)$ holds. Since we do not have any exception $dexcp_p^t$ the correctness of *DMMU* by Lemma 3.4.2 additionally guarantees the following equation:

$$
\begin{aligned}
&din_p^t = \\
&\begin{cases} dtrc^{t-t'}.mem[\langle r\_t(t).pa[31:3]\rangle + 7 : \langle r\_t(t).pa[31:3]\rangle] & \text{if } t \\ dtrc^{t-t'}.mem[\langle ea(c_S)(\tilde{c}_S^i)[31:3] + 7 : ea(c_S)(\tilde{c}_S^i)[31:3]\rangle] & \text{otherwise} \end{cases} \\
&= \begin{cases} c_I^t.M[\langle r\_t(t).pa[31:3]\rangle + 7 : \langle r\_t(t).pa[31:3]\rangle] & \text{if } t \\ c_I^t.M[\langle ea(c_S)(\tilde{c}_S^i)[31:3] + 7 : ea(c_S)(\tilde{c}_S^i)[31:3]\rangle] & \text{otherwise} \end{cases}
\end{aligned}
$$

Since *MU.dout* is given by the output $shift4store$ the access is sign or zero extended and 64 bits wide. The correctness of the $shift4store$ can be found in [Bey05, p.142]. Since all flags in the memory stage are also correct, we finish the lemma. □

Lemma 3.4.9 *Let the CPU correctness invariant without interrupts (Definition 3.3.3) hold in cycle t, let an instruction $i = sI(mem, t)$ in the memory unit complete its aligned access in cycle t. The exception result of this instruction is correct, i.e., we have $corr?(t) \wedge (mr_p^t \vee mw_p^t) \wedge \neg dbusy_p^t \implies$*

$$dexcp_p^t = CA(c_S^i)[dpf]$$

Proof: The proof follows directly from the local correctness of the *DMMU* (see Lemmas 2.3.1 and 3.4.2). □

We can now define a lemma which shows that the memory in cycle t is the same as the memory of the specification, i.e. the correct data is written in the memory in the case of a store operation.

Lemma 3.4.10 *Let the CPU correctness invariant hold in cycle t. The memory part of the invariant still holds in cycle $t+1$, i.e., for all $t \in \mathbb{N}$:*

$$corr?(t) \implies c_I^{t+1}.M = \tilde{c}_S^{sI(M_I, t+1)}.M.$$

Proof: Because of $corr?(t)$, we conclude $c_I^t.M = \tilde{c}_S^{sI(M_I,t)}.M$. If no instruction finishes its memory access in cycle t the equations $sI(M_I, t+1) = sI(M_I, t)$ and $c_I^{t+1}.M = c_I^t.M$ both hold and the claim holds. Let therefore an instruction finish its memory access in cycle t, i.e., we have $(mw_p^t \vee mr_p^t) \wedge \neg dbusy_p^t$. In this case, we have $sI(M_I, t+1) = sI(mem, t)+1$. By applying Lemma 3.4.7 we get $c_I^t.M = \tilde{c}_S^{sI(mem,t)}.M$. We split cases on mw_p^t.

1. Let $\neg mw_p^t$ hold. In this case we have both $c_I^t.M = c_I^{t+1}.M$ and $c_I^{sI(mem,t)}.M = \tilde{c}_S^{sI(mem,t)+1}.M$. Since $corr?(t)$ holds we also finish this case.

2. Let mw_p^t hold. Since mw_p^t is active we have a store instruction in the memory unit. We split cases on $dexcp_p^t$:

   (a) Let $dexcp_p^t$ hold. In this case we have the same proof as in case $\neg mw_p^t$ and we finish this case.

   (b) Let $\neg dexcp_p^t$ hold. The memory stage contains correct data for instruction $sI(mem, t)$, the byte write signals and the data shifted for store are correct and $shift4store$ is correct. The correctness of the $shift4store$ circuit can be found in [Bey05, p.142]. Since we know the correctness of the *DMMU* by Lemma 2.3.1 we are sure that we the correct data is written in the memory. We conclude the proof. □

Note that in both Lemmas 3.4.8 and 3.4.10, we omitted the case of misaligned accesses. In this case we react by simply not accessing the *DMMU* and therefore also not the memory.

3.4.2 Instruction Fetch

In pipelined processors an instruction which has already been fetched could be overwritten by a store instruction which is executed in the memory unit, i.e. we have a RAW hazard. Note that in our case we have more possibilities to have RAW hazards than in [Bey05]. If we work in *user mode* we have RAW hazard not only when data on physical address is overwritten but also when page table entry is overwritten. As a solution to this problem Beyer decided to use a so-called `sync` instruction before the fetch from a modified location. The `sync` instruction stalls the fetch until all previous instructions have left the pipeline. The `sync` instruction has the following properties:

1. In the specification model a `sync` instruction has to be a `nop`.

2. In the implementation model a `sync` instruction prohibits any further fetch of instructions until all instructions which were before the `sync` have left the pipeline.

For our VAMP implementation the instruction `movs2i` *IEEEf*, *R*0 fulfills these criteria. More detail about this case can be found in [Bey05, p. 139–140]. We denote this instruction by `sync`. Since the `sync`?($\mathtt{c_S}$) predicate is a part of the $fetch$ signal such that we prevent fetching of the next instruction after `sync` the second criteria is also fulfilled.

Formally, let *adr* be a physical address. If the instruction $I_i = IR(\tilde{c}_S^i)$ is a fetch from *adr* and we have the instruction I_j which writes to *adr* before the instruction I_i then there must be an instruction I_k between I_i and I_j such that I_k is a `sync` instruction.

We formally define the required `sync` instruction before modified fetches.

Definition 3.4.11 *We call the computation of an assembly code synced 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 double word of address ad occurs in* c_S*, i.e., the effective address in the system mode or physical address in the user mode matched ad and either a store or a floating point store operation occur. Let*

$$r\_tr = decodeitr(c_S.SPR[PTO][19:0], c_S.SPR[PTL][19:0], c_S.M, 1, ea(c_S)).$$

Formally, we define $write_{ad}$ *to double word of address ad as:*

$$\begin{aligned} &write_{ad} :\Longleftrightarrow\ (\texttt{store?}(c_S) \vee \texttt{fstore?}(c_S)) \wedge \\ &(t \wedge (r\_tr.pa[31:3] = ad[31:3]) \vee \neg t \wedge (ea(c_S)[31:3] = ad[31:3])) \end{aligned}$$

Note that since $write_{ad}$ *is a predicate on a configuration, we can use Definition 1.1.9. Since we distinguish two different computations depending on whether we react to interrupts, we 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 under the assumption that we do not have interrupts the following condition holds:

$$\begin{aligned} &synced\_code? :\Longleftrightarrow\ \forall n \in \mathbb{N} : \\ &Let\ ptea = (\langle \tilde{c}_S^n.SPR[PTO][19:0] \circ 0^{12} \rangle + \langle c_S^n.DPC \circ 0^2 \rangle) \bmod 2^{32}, \\ &\quad r\_tr = decodeitr(\tilde{c}_S^n.SPR[PTO][19:0], \\ &\qquad\qquad \tilde{c}_S^n.SPR[PTL][19:0], \tilde{c}_S^n.M, 1, ea(c_S^n))\ \ in \end{aligned}$$

$$\begin{aligned} &(\tilde{c}_S.\exists^{last}_{write_{\tilde{c}_S^n.DPC}} \wedge \neg \tilde{c}_S^n.SPR[MODE][0]\ \Longrightarrow \\ &\qquad \exists m \in]\tilde{c}_S.last_{write_{\tilde{c}_S^n.DPC}} : n[:\quad \texttt{sync?}(IR_S(\tilde{c}_S^m)))\wedge \end{aligned}$$

$$\begin{aligned} &(\tilde{c}_S.\exists^{last}_{write_{r\_tr.pa}} \wedge \tilde{c}_S^n.SPR[MODE][0]\ \Longrightarrow \\ &\qquad \exists m \in]\tilde{c}_S.last_{write_{r\_tr.pa}} : n[:\quad \texttt{sync?}(IR_S(\tilde{c}_S^m)))\wedge \end{aligned}$$

$$\begin{aligned} &(\tilde{c}_S.\exists^{last}_{write_{ptea}} \wedge \tilde{c}_S^n.SPR[MODE][0]\ \Longrightarrow \\ &\qquad \exists m \in]\tilde{c}_S.last_{write_{ptea}} : n[:\quad \texttt{sync?}(IR_S(\tilde{c}_S^m))) \end{aligned}$$

From the view of an assembly programmer, interrupts are in general non-deterministic, e.g., timer-interrupts. This non-determinism is represented by the sequence of external interrupts ext_S*, which is a parameter of a specification computation. Note that we define* ext_S *in Section 3.5.*

$$synced\_code\_i?(ext_S) :\Longleftrightarrow\ \forall n \in \mathbb{N} : synced\_code?[c_S^n]$$

The notation $[c_S^n]$ in this case denote starting configuration reached by the next state function with interrupts. It means that for a machine with

interrupts we only need synced code without interrupt, but from all starting configurations reached by the next state function with interrupts.

Note that in our case the predicate *synced_code*? has the same conditions as before in [Bey05] in case when we work in the *system mode*. In *user mode* we must prevent write access not only to the physical address from which we want to fetch but also page table entry address in the page table from which we get the page table entry in order to compute this physical address.

We now prove that the implementation of the instruction fetch is correct. With the help of the next lemma we are independent of the delayed PC architecture.

Lemma* 3.4.12 *The address forwarding for instruction fetch is correct, i.e., we have for all* $t \in \mathbb{N}$*:*

$$corr?(t) \wedge \neg stall.1^t \implies adr\_i(c_I^t) = \tilde{c}_S^{sI(fetch,t)}.DPC.$$

Proof: We omit the proof because this lemma was taken without changes.

Since we now have correctness of the address adr_i used for instruction fetch the following lemma does not depend on the delayed PC architecture.

Lemma 3.4.13 *Let the VAMP fulfill the correctness invariant in some cycle* t *and let the sync condition on the assembler code hold. The instruction register part of the correctness invariant then holds in the next cycle. I.e., for* $i := sI(dec, t+1)$ *we have for all* $t \in \mathbb{N}$*:*

$$\begin{aligned} synced\_code? \wedge corr?(t) \wedge i \geq 0 \implies \\ (c_I^{t+1}.S1.IR = IR_S(\tilde{c}_S^i) \wedge \\ c_I^{t+1}.S1.imal = imal(\tilde{c}_S^i) \wedge \\ c_I^{t+1}.S1.ipf = CA(\tilde{c}_S^i)[ipf]) \end{aligned}$$

In order to prove this lemma we have to define some intermediate predicates and prove some auxiliary lemmas. If we want to use the correctness results of the local *MMU* as for the *IMMU* we have to give not only sequences of the hardware configuration, inputs, and outputs but also a correct memory sequence. The hardware configuration, inputs, and outputs sequences for the *IMMU* are similar to the hardware configuration, inputs, and outputs sequences for the *DMMU*. Note that the local correctness of the *MMU* has the assumption $m\_ack\_write(trc)$ (during a processor request the memory must be stable while busy is on). Therefore we cannot define a memory mapping in the same way as it was introduced for the *DMMU* because during an request to the *IMMU* the *DMMU* can change the memory.

The following solution will suffice. In order to establish the assumption $m\_ack\_write(trc)$ we construct the memory which is not changed during the whole request to the *IMMU* and is equal to the memory in the last

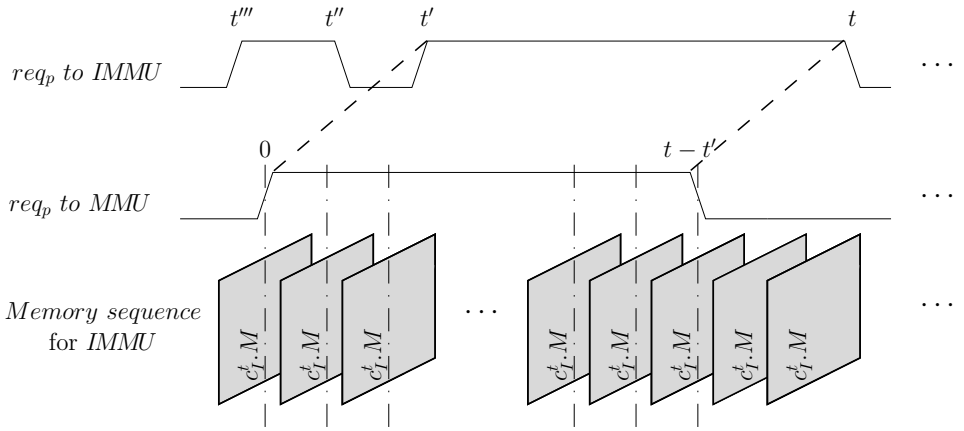

Figure 3.8: Memory Sequence for the *IMMU*

cycle of the request to the *IMMU*. It means that during the whole processor request to the *IMMU* we will use only memory from the last cycle of processor request to the *IMMU*. Figure 3.8 shows memory sequence and construction of the processor request for the *IMMU*.

We also have to define five components that constitute a local *MMU* computation for the *IMMU*. As for the *DMMU* these five components induce a trace *itrc* for the *IMMU*. Formally, we define:

- Memory sequence
$$itrc.mem = \lambda_{t'' \in \mathbb{N}}.c_I^t.M$$

- Inputs sequence for inputs from the processor
$$itrc.in_p = \lambda_{t'' \in \mathbb{N}}. \begin{cases} iinputs_p^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$
where X is an arbitrary $iinputs_p$ such that imr_p is always inactive.

- Inputs sequence for inputs from the memory system
$$itrc.in_m = \lambda_{t'' \in \mathbb{N}}. \begin{cases} iinputs_m^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$
where X is an arbitrary $iinputs_m$ such that $ibusy_m$ is always inactive.

- Outputs sequence for outputs to the processor
$$itrc.out_p = \lambda_{t'' \in \mathbb{N}}. \begin{cases} ioutputs_p^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$
where X is an arbitrary $ioutputs_p$ such that $ibusy_p$ and $iexcp_p$ are always inactive.

- Outputs sequence for outputs to the memory system

$$itrc.out_m = \lambda_{t'' \in \mathbb{N}}. \begin{cases} iinputs_m^{t''+t'} & \text{if } t' + t'' \leq t \\ X & \text{otherwise} \end{cases}$$

  where X is an arbitrary $iinputs_m$ such that imr_m is always inactive and $iaddr_m = iaddr_m^{t''+1}$.

Note that we also as for the *DMMU* have to show that this computation is a computation of the local *MMU*. It also is easy to show that these definitions constitute a valid local *MMU* computation. In order to show that the assumptions for the local *MMU* correctness hold we prove a lemma which shows that assumption $p\_req\_is\_stable(itrc)$ holds for the *IMMU*.

Lemma 3.4.14 *All inputs to the IMMU are stable when the IMMU is busy in the same cycle:*

$$p\_req\_is\_stable(itrc)$$

Proof: Note that we give the proof only for the time when we have the processor request, i.e. from cycle 0 to cycle $t - t'$ (see Figure 3.8). After cycle $t - t'$ the predicate $p\_req\_is\_stable(itrc)$ trivially holds because of the trace construction.

Let t'' denote the cycle which we get by skolemization of the universal quantifier in the predicate $p\_req\_is\_stable(itrc)$. Since any request to the *IMMU* is a read request and $imr_p^{t''} = fetch^{t''} \wedge \neg c_I^{t''}.S1.imal \vee irollback^{t''}$ we only have to prove that:

$$(fetch^{t''} \wedge \neg c_I^{t''}.S1.imal \vee irollback^{t''}) \wedge ibusy_p^{t''} \implies iinputs_p^{t''} = iinputs_p^{t''+1}$$

We split the input stability into three groups. The first is $iadr_p$, the second is imr_p and the last is $ipto_p$, $iptl_p$, and $imode_p$.

Subproof for $iadr_p$. Note that $iadr_p = adr\_i(c_I)[31:3]$ and since we also need to have stability of the last three bits later for the proof of the imr_p stabilizing we prove that the whole $adr\_i(c_I)$ is stabilized. We now split cases on $irollback^{t''}$:

1. Let $irollback^{t''}$ hold. Since $ibusy_p^{t''}$ holds we have $irollback^{t''+1}$. Since $irollback$ holds in both cycles t'' and $t'' + 1$ we have $adr\_i(c_I^{t''}) = mPC^{t''}$ and $adr\_i(c_I^{t''+1}) = mPC^{t''+1}$ according to Figure 3.5. Because $irollback^{t''}$ is active we do not update the register mPC between cycles t'' and $t'' + 1$. This finishes this case.

2. Let $\neg irollback^{t''}$ hold. In this case we have to prove $adr\_i(c_I^{t''}) = adr\_i(c_I^{t''+1})$. Since $fetch^{t''}$ implies $\neg \texttt{rfe}?(c_I^{t''}.S1.IR)$ we have

$$adr\_i(c_I^{t''}) = \begin{cases} c_I^{t''}.PC' & \text{if } c_I^{t''}.S1.full \wedge \neg \texttt{rfe}?(c_I^{t''}.S1.IR) \\ c_I^{t''}.DPC & \text{otherwise} \end{cases}$$

We compute $adr\_i(c_I^{t''+1})$ as follows

$$adr\_i(c_I^{t''+1}) = \begin{cases} c_I^{t''+1}.SPR[EDPC] & \text{if } c_I^{t''+1}.S1.full \wedge \texttt{rfe}?(c_I^{t''+1}.S1.IR) \\ c_I^{t''+1}.PC' & \text{if } c_I^{t''+1}.S1.full \wedge \neg \texttt{rfe}?(c_I^{t''+1}.S1.IR) \\ c_I^{t''+1}.DPC & \text{otherwise} \end{cases}$$

Since signal $ibusy_p^{t''}$ is active we have $\neg ue.0^{t''}$. Since $c_I.S1.IR$ is only changed when signal $ue.0$ is active we have $c_I^{t''+1}.S1.IR = c_I^{t''}.S1.IR$. Note also that since signal $ue.0^{t''}$ is inactive we have $S1.full^{t''+1} = stall.1^{t''}$ and from the construction of the signal $stall.1^{t''}$ we conclude that $stall.1^{t''} \implies S1.full^{t''}$ trivially. We now split cases on $stall.1^{t''}$.

(a) If $stall.1^{t''}$ holds then we have $adr\_i(c_I^{t''}) = c_I^{t''}.PC'$ and $adr\_i(c_I^{t''+1}) = c_I^{t''+1}.PC'$. Since PC' is changed only when $ue.1$ is active we have:

$$adr\_i(c_I^{t''+1}) = c_I^{t''+1}.PC' = c_I^{t''}PC' = adr\_i(c_I^{t''})$$

and finish this case.

(b) Let $\neg stall.1^{t''}$ hold. Note that from the construction $\neg stall.1^{t''} \implies \neg \texttt{rfe}$ trivially holds. We now split cases on $S1.full^{t''}$.

i. If $S1.full^{t''}$ holds then $adr\_i(c_I^{t''}) = c_I^{t''}.PC'$ and $adr\_i(c_I^{t''+1}) = c_I^{t''+1}.DPC$. Since we compute $c_I^{t''+1}.DPC$ by:

$$c_I^{t''+1}.DPC = \begin{cases} c_I^{t''}.PC' & \text{if } ue.1 \\ c_I^{t''}.DPC & \text{otherwise} \end{cases}$$

and $ue.1$ is active we have:

$$adr\_i(c_I^{t''+1}) = c_I^{t''+1}.DPC = c_I^{t''}PC' = adr\_i(c_I^{t''})$$

and finish this case.

ii. If $\neg S1.full^{t''}$ holds then $adr\_i(c_I^{t''}) = c_I^{t''}.DPC$ and $adr\_i(c_I^{t''+1}) = c_I^{t''+1}.DPC$. Since signal $ue.1$ is inactive we have

$$adr\_i(c_I^{t''+1}) = c_I^{t''+1}.DPC = c_I^{t''}DPC = adr\_i(c_I^{t''})$$

and finish the part of the lemma for $iadr_p$.

Subproof for imr_p. We have to prove that $imr_p^{t''} = imr_p^{t''+1}$. In case when $irollback^{t''}$ is active $imr_p^{t''} = imr_p^{t''+1}$ follows directly from the construction of the stabilize circuit for *IMMU* (Figure 3.5) since we have $ibusy_p^{t''}$. Let $\neg irollback^{t''}$ hold. Of course, $irollback^{t''+1}$ also is inactive. Since we have already proved stability of the address we have $c_I^{t''}.imal = c_I^{t''+1}.imal$ and we only have to prove that $1 = fetch^{t''} = fetch^{t''+1}$. We now split cases on $stall.1^{t''}$.

1. If $stall.1^{t''}$ holds then as we know $S1.full^{t''}$ and $S1.full^{t''+1}$ are both active. Since we know from the previous part of proof that $c_I^{t''+1}.S1.IR = c_I^{t''}.S1.IR$ according to the definition of the signal $fetch$ we only have to prove that registers *PTO*, *PTL*, and *MODE* are valid. Since validity of a register is changed from 1 to 0 in case of $S1.full \wedge \neg stall.1$ we conclude this case.

2. Let $\neg stall.1^{t''}$ hold. We now split cases on $S1.full^{t''}$.

   (a) Let $S1.full^{t''}$ hold. In this case we also have that $c_I^{t''+1}.S1.IR = c_I^{t''}.S1.IR$ and we only have to show validity of registers *PTO*, *PTL*, and *MODE*. Since in cycle t'' we do not have any instruction which changes the validity of registers *PTO*, *PTL*, and *MODE* we finish this case.

   (b) Let $\neg S1.full$ hold. Since the validity of an register can changed from 1 to 0 only when $S1.full$ we finish the claim for imr_p.

Subproof for $ipto_p$, $iptl_p$, **and** $imode_p$. Note that $ipto_p$, $iptl_p$, and $imode_p$ are read directly from the *SPR* file. We only show the stability of $ipto_p$ because the proofs for the other two registers are similar. We split cases on $irollback^{t''}$.

1. Let $irollback^{t''}$ hold. By induction we can prove that $irollback^{t''} \implies c_I.ROBcount = 0 \wedge \neg c_I.S1.full$. Thus $c_I.ROBcount = 0$; since the content of registers is changed only in stage writeback and iff $c_I.ROBcount \neq 0$ we finish this case.

2. If the content of register $c_I^{t''}.SPR[PTO]$ is changed this register is not valid. Thus we have contradiction because we have $c_I.SPR[PTO].valid$. □

We now prove that the local *MMU* correctness holds for the trace *itrc*.

Lemma 3.4.15 *Local MMU correctness for the trace itrc holds:*

$$\forall t', t \in \mathbb{N} : is\_req\_proc(t', t, itrc) \wedge corr?(t) \wedge \neg irollback^t \wedge synced\_code? \wedge adr\_i(c_I^t) = c_S^{sI(fetch,t)}.DPC \implies mmu\_guarantee(itrc)$$

In order to prove this lemma we need some additional lemmas which are partially taken from [Bey05].

Lemma* 3.4.16 *Each memory instruction, which terminated without exception is committed, i.e.:*

$$\begin{aligned}\forall t, i \in \mathbb{N} : \texttt{mem?}(c_S^i) \wedge \neg imal(c_S^i) \wedge \neg dmal(c_S^i) \wedge \\ \neg ipf(c_S^i) \wedge \neg dpf(c_S^i) \wedge sI(wb,t) > i \implies \\ sI(M_I,t) > i\end{aligned}$$

Lemma* 3.4.17 *If a content of a double word memory cell is not equal in two different cycles then there exists an instruction in between which changes this content, i.e.:*

$$\begin{aligned}\forall i \in \mathbb{N}, i' \in \mathbb{Z}_{\geq i}, adr \in \mathbb{B}^{32} \ : \ c_S^i.M[adr[31:3]] \neq c_S^{i'}.M[adr[31:3]] \implies \\ \exists j \in [i : i'[\ : \ write_{adr}(c_S^j) \wedge \texttt{mem?}(c_S^j) \wedge \neg imal(c_S^j) \wedge \\ \neg dmal(c_S^j) \wedge \neg ipf(c_S^j) \wedge \neg dpf(c_S^j)\end{aligned}$$

Lemma 3.4.18 *An instruction which changes data in the memory at an address, which is a page table entry address for another instruction, is terminated, when that other instruction is fetched, i.e.:*

$$\begin{aligned}\forall t \in \mathbb{N} : \ let \ \ ptea = (\left\langle \tilde{c}_S^{sI(fetch,t)}.SPR[PTO][19:0] \circ 0^{12} \right\rangle + \\ \left\langle c_S^{sI(fetch,t)}.DPC \circ 0^2 \right\rangle) \bmod 2^{32} \ in\end{aligned}$$

$$\begin{aligned}synced\_code? \wedge fetch^t \implies \\ \forall i \in \mathbb{Z}_{sI(fetch,t)} \ : \ write_{ptea}(c_S^i) \implies \\ sI(wb,t) > i\end{aligned}$$

Lemma 3.4.19 *An instruction which changes data in the memory at an address which is a physical address for another instruction, is terminated, when that other instruction is fetched, i.e.:*

$$\begin{aligned}\forall t \in \mathbb{N} : \ let \ r\_tr = decodeitr(c_S^{sI(fetch,t)}.SPR[PTO][19:0], \\ c_S^{sI(fetch,t)}.SPR[PTL][19:0], c_S^{sI(fetch,t)}.M, \\ c_S^{sI(fetch,t)}.SPR[MODE][0], ea(c_S^{sI(fetch,t)})) \ in\end{aligned}$$

$$
\begin{aligned}
&synced\_code? \wedge \neg ibusy_p^t \wedge stall.1^t \wedge fetch^t \wedge \neg irollback^t \implies \\
&\quad \forall i \in \mathbb{Z}_{sI(fetch,t)} \ : \ (write_{r\_tr.pa}(c_S^i) \wedge \\
&\qquad c_S^{sI(fetch,t)}.SPR[MODE][0] \vee \\
&\qquad write_{c_S^{sI(fetch,t)}.DPC}(c_S^i) \wedge \\
&\qquad \neg c_S^{sI(fetch,t)}.SPR[MODE][0]) \implies \\
&\qquad\quad sI(wb,t) > i
\end{aligned}
$$

We omit the proofs of Lemmas 3.4.18 and 3.4.19; similar properties (without address translation) have been proven in [Bey05].

Proof: [Lemma 3.4.15] Since on instruction fetch we produce for the *IMMU* only read accesses the assumption $p\_mr\_mw\_mutexc(itrc)$ always holds. Lemma 3.4.14 proves that the predicate $p\_req\_is\_stable(itrc)$ also holds. Therefore the predicate $good\_p\_interface(itrc)$ holds. Assumption of the memory access for the *IMMU*: $m\_ack\_write(itrc)$, $m\_write\_consist(itrc)$, and $m\_liveness(itrc)$ are fulfilled because of Definition 1.5.1 up to the end of the processor request, i.e. up to cycle $t-t'$. Since imr_p is inactive after the request to the *IMMU* the control of the *IMMU* is in state *idle*. Then we conclude that imr_m is also inactive. Since imr_m is inactive all these predicates trivially hold. In order to show correctness of the *IMMU*, we only have to prove that the predicate $m\_read\_consist(itrc)$ also holds. Let t_s and t_e denote start and end time of an memory request such that the following inequality holds:

$$0 \leq t_s \leq t_e \leq t - t'$$

Since during the whole processor request memory sequence in $itrc$ is stable according to our construction of $itrc$ trace equation $itrc(t_e).mem = itrc(t_e+1).mem$ trivially holds. Therefore we only have to prove that:

$$itrc(t_e).in_m.din = itrc(t_e).mem[itrc(t_s).out_m.adr]$$

With the help of Definitions 1.5.1 and 1.5.2 we rewrite this equation as follows:

$$C_I^{t'+t_e}.M[iadr_m^{t'+t_e}] = itrc(t_e).mem[itrc(t_s).out_m.adr]$$

Since from Lemmas 2.1.19 and 2.3.4 we know that inputs to the memory are stable and we rewrite further:

$$C_I^{t'+t_e}.M[iadr_m^{t'+t_e}] = C_I^t.M[iadr_m^{t'+t_e}]$$

Since the predicate $corr?(t)$ holds for all cycles up to t it suffices to show

$$c_S^{sI(M_I,t'+t_e)}.M[iadr_m^{t'+t_e}] = c_S^{sI(M_I,t)}.M[iadr_m^{t'+t_e}]$$

In case $t' + t_e = t$ the equation trivially holds. In other case in order to find contradiction we assume that

$$c_S^{sI(M_I,t'+t_e)}.M[iadr_m^{t'+t_e}] \neq c_S^{sI(M_I,t)}.M[iadr_m^{t'+t_e}]$$

From Lemma 3.4.17 follows that there exist an instruction j which has overwritten data at the address $iadr_m^{t'+t_e}$, i.e.:

$$\exists j \in [t' + t_s : t[\ : \ write_{iadr_m^{t'+t_e}}(c_S^j) \wedge \texttt{mem}?(c_S^j) \wedge \neg imal(c_S^j) \wedge \\ \neg dmal(c_S^j) \wedge \neg ipf(c_S^j) \wedge \neg dpf(c_S^j)$$

In Lemma 2.3.10 we proved that our local *MMU* can access the memory during a request only at two addresses in *user* mode pagetable entry address and physical address and in *system* mode only at physical address. Since accesses at physical addresses in both modes we can have only if $t' + t_e = t$ we work in *user* mode and $iadr_m^{t'+t_e}$ is equal to the page table entry address. We also know that

$$sI(M_I, t' + t_e) \leq j \leq sI(M_I, t) \leq sI(fetch, t) = sI(fetch, t' + t_e)$$

Therefore since *synced_code*? and $fetch^{t'+t_e}$ both hold we get from Lemma 3.4.18 that $sI(wb, t' + t_e) > j$. According to the Lemma 3.4.16 we have that $sI(M_I, t' + t_e) > j$. Therefore we have found a contradiction $sI(M_I, t' + t_e) > j \geq sI(M_I, t' + t_e)$.

Therefore we know that the predicate *good_m_interface*(*dtrc*) also holds. So we have already shown correctness of all assumptions for the *IMMU*. Therefore we now know the correctness of the *IMMU* access. □

Note also that in case if we have last cycle of the request to the *IMMU* then we can easily reconstruct from the last cycle of an *IMMU* request the whole request.

Lemma 3.4.20 *If a fetch to the IMMU terminates in cycle t then there exists a cycle $t' \leq t$ when this access is started. Formally, we have:*

$$\forall t \in \mathbb{N} : imr_p^t \wedge \neg ibusy_p^t \wedge \neg irollback^t \implies \\ \exists t' \in \mathbb{Z}_{\leq t} : is\_req\_proc(t', t, itrc)$$

The proof is done by induction on t' and omitted here.

We now can prove correctness on the instruction fetch, i.e. Lemma 3.4.13.
Proof: [Lemma 3.4.13] Let *synced_code*?, *corr*?(t), and $i \geq 0$ hold. If no fetch occurs in cycle t, i.e., $\neg ue.0^t$, we set $i = sI(dec, t)$, $c_I^{t+1}.S1.IR = c_I^t.S1.IR$, $c_I^{t+1}.S1.imal = c_I^t.S1.imal$, $c_I^{t+1}.S1.ipf = c_I^t.S1.ipf$, and the proof is finished because of the instruction register part of *corr*?(t) holds. Hence, we assume that a fetch occurs, i.e., $\neg ibusy_p^t \wedge \neg stall.1^t \wedge fetch^t \wedge \neg irollback^t$. We define $i = sI(dec, t)+1$ and $i = sI(fetch, t)$. By Lemma 3.4.12 we obtain

that $adr\_i(c_I^t) = c_S^i.DPC$ holds. If $c_S^i.DPC$ is misaligned, the claim holds trivially. Hence, we have to show the above claim for the instruction register itself and page fault on instruction fetch.

Let us have an aligned $adr\_i(c_I^t)$, i.e., $\tilde{c}_S^i.DPC \bmod 4 = 0$. Note that the instruction port of the CPU_I interface is accessed on address $iadr_p^t = adr\_i(c_I^t)$ div 8 and depending on $adr\_i(c_I)^t \bmod 8$, either the upper or lower word is selected as input to the instruction register. Because of $\neg ibusy^t \wedge fetch^t \wedge \neg irollback^t$ and Lemma 3.4.20 we have a cycle t' when the request to the *IMMU* is started, i.e., we have $is\_req\_proc(t', t, itrc)$.

For the following equations we define short notation as follows:

$$pa(c_I) = decodeitr(c_I^t.SPR[PTO][19:0], c_I^t.SPR[PTL][19:0], c_I^t.M, 0, c_I^t.DPC).pa,$$

$$pa(c_S) = decodeitr(\tilde{c}_S^i.SPR[PTO][19:0], \tilde{c}_S^i.SPR[PTL][19:0], \tilde{c}_S^i.M, 0, \tilde{c}_S^i.DPC).pa.$$

We now consider two cases:

1. The next implications both hold.

$$\begin{aligned}
&(\neg\tilde{c}_S^i.SPR[MODE][0] \implies \\
&\quad c_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] = c_S^i.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC]) \wedge \\
&(\tilde{c}_S^i.SPR[MODE][0] \implies \\
&\quad c_S^{sI(M_I,t)}.M[pa(c_S) + 3 : pa(c_S)] = c_S^i.M[pa(c_S) + 3 : pa(c_S)] \wedge \\
&\quad c_S^{sI(M_I,t)}.M[ptea(c_S) + 3 : ptea(c_S)] = c_S^i.M[ptea(c_S) + 3 : va(c_S)])
\end{aligned}$$

where

$$ptea(c_S) = (\langle \tilde{c}_S^i.SPR[PTO][19:0] \circ 0^{12} \rangle + \langle c_S^i.DPC \circ 0^2 \rangle) \bmod 2^{32}.$$

Note that we have already proved in Lemma 2.3.10 that *MMU* access the memory during any untranslated processor request at most one time on address $\tilde{c}_S^i.DPC$. If we have a translated processor request then we access the *MMU* at most two times. The first time we access the page table entry address and the second the physical address. By applying Lemma 3.4.15 we conclude the correctness of the *IMMU*, i.e. $mmu\_guarantee(itrc)$. Therefore by using Definition 1.5.2 of a correct memory interface and $corr?(t)$ in the case if we do not have

any exception ($iexcp_p^t$) we obtain:

$$c_I^{t+1}.S1.IR =$$

$$\begin{cases} c_I^t.M[8 \cdot iadr_p^t + 7 : 8 \cdot iadr_p^t + 4] & \text{if } adr\_i(c_I^t) \bmod 8 = 4 \wedge \neg imode_p^t \\ c_I^t.M[8 \cdot pa(c_I) + 7 : 8 \cdot pa(c_I) + 4] & \text{if } adr\_i(c_I^t) \bmod 8 = 4 \wedge imode_p^t \\ c_I^t.M[8 \cdot iadr_p^t + 3 : 8 \cdot iadr_p^t + 0] & \text{if } adr\_i(c_I^t) \bmod 8 \neq 4 \wedge \neg imode_p^t \\ c_I^t.M[8 \cdot pa(c_I) + 3 : 8 \cdot pa(c_I) + 0] & \text{if } adr\_i(c_I^t) \bmod 8 \neq 4 \wedge imode_p^t \end{cases}$$

$$= \begin{cases} c_I^t.M[pa(c_S) + 3 : pa(c_S)] & \text{if } \tilde{c}_S^i.SPR[MODE][0] \\ c_I^t.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] & \text{otherwise} \end{cases}$$

$$= \begin{cases} \tilde{c}_S^{sI(M_I,t)}.M[pa(c_S) + 3 : pa(c_S)] & \text{if } \tilde{c}_S^i.SPR[MODE][0] \\ \tilde{c}_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] & \text{otherwise} \end{cases}$$

$$= \begin{cases} \tilde{c}_S^i.M[pa(c_S) + 3 : pa(c_S)] & \text{if } \tilde{c}_S^i.SPR[MODE][0] \\ \tilde{c}_S^i.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] & \text{otherwise} \end{cases}$$

The first equation holds because of the memory unit construction. The second equation holds because the predicate $mmu\_guarantee(itrc)$ holds and $adr\_i(c_I^t) = c_S^i.DPC$. Since the predicate $corr?(t)$ holds we get the third equation. The last equation holds, because the above two implications hold.

For page fault on instruction fetch with the help of Lemma 3.4.15 we have

$$\begin{aligned} c_I^{t+1}.CA[ipf] &= ipf(c_I^t) \wedge c_I^t.SPR[MODE][0] \\ &= ipf(c_S^i) \wedge \tilde{c}_S^i.SPR[MODE][0] \end{aligned}$$

where $ipf(c_I^t)$ is page fault on fetch in the implementation in cycle t and $ipf(c_S^i)$ is the page fault on fetch in the specification. Note that both are equal because of *IMMU* correctness (Lemma 3.4.15) and because of $corr?(t)$. Therefore

$$IR_S(c_S^i) = \begin{cases} c_S^{sI(M_I,t)}.M[pa(c_S) + 3 : pa(c_S)] & \text{if } \tilde{c}_S^i.SPR[MODE][0] \\ c_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC + 3 : \tilde{c}_S^i.DPC] & \text{otherwise} \end{cases}$$

holds and the claim is finished.

2. Assume the implications

$$(\neg\tilde{c}_S^i.SPR[MODE][0] \implies c_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC+3:\tilde{c}_S^i.DPC] = c_S^i.M[\tilde{c}_S^i.DPC+3:\tilde{c}_S^i.DPC]) \wedge$$
$$(\tilde{c}_S^i.SPR[MODE][0] \implies c_S^{sI(M_I,t)}.M[pa(c_S)+3:pa(c_S)] = c_S^i.M[pa(c_S)+3:pa(c_S)] \wedge$$
$$c_S^{sI(M_I,t)}.M[ptea(c_S)+3:ptea(c_S)] = c_S^i.M[ptea(c_S)+3:va(c_S)])$$

do not both hold.

Since proofs of both parts of this equation are similar we only show the proof for the case when $\tilde{c}_S^i.SPR[MODE][0]$ is inactive. Therefore in order to find contradiction we assume that

$$c_S^{sI(M_I,t)}.M[\tilde{c}_S^i.DPC+3:\tilde{c}_S^i.DPC] \neq c_S^i.M[\tilde{c}_S^i.DPC+3:\tilde{c}_S^i.DPC]. \tag{3.34}$$

Note that since instruction i is fetched into the pipeline in cycle t, we can prove by induction easily that $i \geq sI(M_I,t)$ holds. According to Lemma 3.4.17 there exists an instruction $j \in [sI(M_I,t):i[$ such that the following predicate holds

$$write_{c_S^{sI(fetch,t)}.DPC}(c_S^j) \wedge \texttt{mem?}(c_S^j) \wedge \neg imal(c_S^j) \wedge$$
$$\neg dmal(c_S^j) \wedge \neg ipf(c_S^j) \wedge \neg dpf(c_S^j)$$

From Lemma 3.4.19 - which uses *synced_code*? - and since $j < i$ we conclude that $sI(wb,t) > j$. If it is true then according to Lemma 3.4.16 we obtain that $sI(M_I,t) > j$ what contradict to $sI(M_I,t) \leq j$. In a similar way we get the same contradiction in case of user mode for both addresses: page table entry address and physical address and we finish the proof. □

3.5 Correctness with Interrupts

In this section we extend correctness of the VAMP with the correctness proof of interrupts. Since correctness of the internal interrupts was already established in [Bey05] we will focus on external interrupts. The proofs have only a slight difference, namely the predicate *JISR* is now formulated over internal and external interrupts.

In order to specify external interrupts we introduce a function $ext(c_S)$ which represents external interrupts in the specification. This function maps a number of instruction i to the bitvector of external interrupts.

$$ext_S(i) = \lambda_{x\in\mathbb{Z}_{19}}.\exists t \in \mathbb{N}: (ext^t[x] \wedge (writeback^t \vee JISR(c_I^t)) \wedge sI(inst,t) = i)$$

Note that the VAMP processor samples external interrupt signals only during the writeback stage, i.e. it does not sample them all. Every instruction is only for one cycle in the writeback stage.

Since in the VAMP implementation all interrupts are triggered during writeback it is easy to prove correctness of the implementation with respect to the specification of external interrupts. All instructions also leave the pipeline in order because of the reorder buffer, i.e. preciseness of interrupts with respect to the register files is easy to show. The following lemmas we give without proofs since all of proofs are similar to the proofs for the VAMP without address translation and can be found in [Bey05, p. 154–158].

Lemma* 3.5.1 *VAMP correctness with interrupts holds up to the cycle of the first interrupt, i.e., for all $t \in \mathbb{N}$:*

$$(synced\_code\_i?(ext_S) \wedge \forall t' \in \mathbb{Z}_t : \neg JISR(c_I^{t'})) \implies corr?(t) \wedge corr\_i?(t)$$

Lemma* 3.5.2 *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., for all $t \in \mathbb{N}$:*

$$(synced\_code? \wedge \forall t' \in \mathbb{Z}_t : \neg JISR(c_I^{t'})) \wedge JISR(c_I^t) \implies \tilde{c}_S^{sI(wb,t)+1}.M = \tilde{c}_S^{sI(M_I,t)}.M$$

Lemma* 3.5.3 *In case of an interrupt, no write to the data memory is currently in progress, for all $t \in \mathbb{N}$:*

$$(\forall t' \in \mathbb{Z}_t : \neg JISR(c_I^{t'})) \wedge JISR(c_I^t) \implies \neg mw_p^t$$

Lemma* 3.5.4 *Let the first interrupt occur in cycle t. The memory part of $corr\_i?(t+1)$ then holds, i.e., we have for all $t \in \mathbb{N}$:*

$$(synced\_code\_i?(ext_S) \wedge \forall t' \in \mathbb{Z}_t : \neg JISR(c^{t'})) \wedge JISR(c_I^t) \implies c_I^{t+1}.M = c_S^{sI(inst,t+1)}.M$$

Lemma* 3.5.5 *VAMP correctness with interrupts also holds in the cycle immediately after the first interrupt. Formally, we have for all $t \in \mathbb{N}$:*

$$(synced\_code\_i? \wedge \forall t' \in \mathbb{Z}_{t-1} : \neg JISR(c_I^{t'})) \implies corr\_i?(t)$$

Proposition* 3.5.6 *Initially and in the cycle after an interrupt, the VAMP is an initial state according to Definition 3.2.1. Formally, we have for all $t \in \mathbb{N}$:*

$$(t = 0 \vee JISR(c_I^{t-1})) \implies init?(c^t)$$

3.5.1 Overall Correctness

We now show overall correctness of the VAMP processor with address translation. Since we have localized all changes in module proofs we can use the top level of the previous VAMP proof without logical changes. We have only to do technical changes since we have introduced external interrupts as a new parameter. In order to prove the main correctness theorem we first have to prove an auxiliary lemma which states that correctness in some cycle $t' + 1$ can be reduced to the correctness in cycle of the last interrupt before t'.

Lemma* 3.5.7 *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 for all $t \in \mathbb{N}$:*

$$synced\_code\_i? \wedge corr\_i?(t) \wedge (t = 0 \vee JISR(c_I^{t-1})) \implies \\ \forall t' \in \mathbb{Z}_{\geq t} : (\forall t'' \in [t : t'[: \neg JISR(c_I^{t''})) \implies corr\_i?(t' + 1)$$

The proof can be found in [Bey05, p. 159]. Finally we can introduce the main theorem of the VAMP correctness with address translation.

Theorem 3.5.8 *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 ($\mathbf{t = 0}$): Since $corr\_i?(0)$ holds by definition we finish this case.
Induction step ($\mathbf{t \to t + 1}$): We have to show $corr\_i?(t + 1)$. We now split cases on $c_I.\exists^{last}_{JISR}(t)$.

1. Let $\neg c_I.\exists^{last}_{JISR}(t)$ hold. According to the definition of $\exists^{last}$ we have $\forall k \in \mathbb{Z}_t : \neg JISR(c_I^k)$. Since we have already proved that $corr\_i?(0)$ holds in the base case, we can apply Lemma 3.5.7 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 using Definition 1.1.9 for $last$, 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 conclude $corr\_i?(t + 1)$ by applying Lemma 3.5.7 to cycles $l + 1$ and t. This finally finishes the main theorem and proves that the VAMP with address translation does not contain any functional error. □

Chapter 4

Conclusion

4.1 Summary

In this thesis we reported about the formal verification of a complex processor supporting address translation and external interrupts. Our work was based on the following previous work: paper and pencil correctness of memory management units [Hil05], formal verification of the Tomasulo algorithm [Krö01] and formal verification of the VAMP processor without address translation [Bey05]. The VAMP processor is an 32-bits RISC processor with a Tomasulo scheduler, a memory unit with cache memory interface, fixed point unit, and three floating point units. The result of this thesis are published in [DHP05].

In this thesis, we have presented implementation and the formal specification of a simple memory management unit and formally verified its correctness. The correctness proof of a *MMU* alone is simple, but depends on nontrivial operating conditions. As a next step, we have integrated the *MMU* into the memory unit of the VAMP processor [Bey05, Krö01, JK00, BBJ$^+$02, BJK$^+$03, BJK$^+$05, Jac02a] for the address translation on load/store and into the stage fetch for address translation on instruction fetch.

In addition to the implementation, we also introduced the programmer's model of the VAMP which we use as specification in order to prove correctness of the VAMP. We showed the overall functional correctness of the VAMP implementation with respect to the specification. We focused on the data memory access, self-modifying code, instruction fetch and precise external interrupts.

In Table 4.1 we compare of the different verification effort of proofs which were done in PVS at our institute.

Index	Name of Theory	# Commands	# Lemmas
1	Basics Circuits	5505	134
2	ALU	2202	76
3	Pipelined DLX	1850	53
4	Tomasulo alg.	5847	186
5	VAMP without *MMU*s (w/o Tomasulo alg.)	24753	495
6	Local *MMU*	7399	85
7	VAMP with *MMU*s (w/o Tomasulo alg.)	36785	587

Table 4.1: Comparison of the Verification Effort in PVS.

4.2 Related Work

Note that hardware verification is probably one of most active and successful fields in formal verification. Nowadays there exist several works in the field of formal processor verification with a Tomasulo algorithm as well as without. The VAMP project has been started in 2000 at Saarland University. Using the theorem prover PVS, Beyer et al. showed the correctness of a powerful VAMP microprocessor without virtual memory support [BBJ$^+$02, BJK$^+$03, BJK$^+$05, Bey05, Krö01, JK00]. Sawada and Hunt [SH98, HS99] showed the correctness of an out-of-order processor with precise interrupts and a store buffer for the memory unit. They also decided to use `sync` instruction to deal with self-modifying code. McMillan showed the correctness an out-of-order processor based on Tomasulo algorithm using compositional model checking techniques [McM98]. For this purpose he used the SMV verifier and applied symmetry to reduce the number of assertions that have to be verified. Brock and Hunt report about the formal verification of a very simple, non pipelined processor FM9001 [BHK94]. Aagard et al. presented in [ACHK04] a new approach for formal verification of pipelined processor by using equivalence verification and completion functions. They have used this approach to verify a 32-bit RISC processor with 47 instructions, four-stages, and in-order execution.

But *all* results do not cover virtual memory support. Already Beyer noted in [Bey05] that the version of the VAMP which was presented in [Bey05] 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. *All* results above do not support address translation. We are not aware of previous work on formal verification of processors with *MMU* as well as memory management units itself.

Processor designs with address translation by means of an *MMU* with

translation look aside buffers is presented, e.g., in [HP96a]. These designs are not formally specified or proven correct. Jacob and Mundge provide a survey of address translation mechanism present in modern microprocessors [JM98].

Note that now there is more activity on the field of software verification than hardware verification.

4.3 Future work

Presently we see several main directions for further work. First, we should try to reprove VAMP using powerful automated tools. Second, on the hardware side one wants to verify processors with pipelined *MMU*s, multi level page tables and table look aside buffers.

4.3.1 Automated Methods

The main drawback of interactive proof tools is the strong dependency on the concrete implementation. Even slight modifications in some part of the implementation may cause significant changes of the proofs.

Suppose at a certain step of the proof one discovers a bug in the implementation. In most cases the only possibility to fix a bug is to change the implementation. Therefore, one has to change the part of proof which we have done as well. Thus, the verification process becomes iterative. Schematically:

$$\begin{aligned}&\textbf{obtain a part of proof} \implies\\ &\textbf{discover a subtle bug} \implies\\ &\textbf{change the implementation} \implies\\ &\textbf{change the proof}\end{aligned}$$

This process consumes much time. Note that in our case before the formal verification in the PVS mathematical correctness was done with paper and pencil. Thus, all bugs which we found by formal verification in PVS were bugs which we missed in the paper and pencil proof. Since industrial processors usually do not have a mathematical correctness proof, the number of iterations will increase. Therefore we need to find some automated methods in order to decrease proof time. Using model checking and a light-weight completion function Clarke et al. have shown the correctness of the Tomasulo's algorithm in [BCBZ02]. By using efficient reductions of the equality with uninterpreted functions Velev and Bryant in [VB99] entirely automatically correctness of a set of superscalar DLX implementations [HP96b] with in-order execution, having 2 pipelines with 5 stages each. The work presented in this paper has several simplifications and abstractions. Some examples of such abstractions are: load instructions on fetch was produced by an uninterpreted function and next step computation of the memory was also done

by using an uninterpreted function. Manolios and Srinivasan [MS05] presented results which can be used to speed up runtime of decision procedures for automated tools.

In order to decrease proof time the automated proof tool NuSMV is used in our institute. NuSMV is a model checker with an integrated SAT solver [Tve05a]. Model checking techniques are widely used to verify correctness of hardware systems [BD94, McM98]. The main problem with model checking is the state explosion problem, i.e. the state space grows exponentially with the system size. Abstraction is the necessary approach to apply model checking methods to large systems. The goal of abstraction is to reduce the size of the state space while still being able to show the desired property. Sergey Tverdyshev has already proved with the help of NuSMV and some additional tools the correctness of all basic circuits and most part of ALU of the VAMP [Tve05b].

We now discuss which parts of the processor could also be verified with the automated tools.

- *MU* will be difficult to verify as one module because of the big state space. One can proceed by splitting the *MU* into several parts, *DMMU*, $shift4store$, *compadr*, etc. Note that correctness proof of the local *MMU* with the automated methods should be possible to do for the current model of *MMU* after using data abstractions. Our implementation of the *MMU* takes several bitvectors as an input. For example we describe the data abstraction for the page table length *ptl*. Since *ptl* is a bitvector of length 20 we have 2^{20} possible states only for *ptl*. We can reduce the state space to two states for the *ptl* if we use data abstraction. It will suffice since we use *ptl* only once, when we compare it with *pto*, i.e. two states will be enough in order to distinguish the result of the compare operation. Of course, such abstraction need to be done for all inputs where it is possible. We can substitute 20-bits adders for 32-bits adders which is used in order to compute page table entry address. It also reduce the state space. After such a sequence of reducing steps the model checker should hopefully prove the correctness of the *MMU*.

- The same approach can be used to establish the correctness of the *IMMU* on the instruction fetch.

- Proofs of *FPU*s already involve automated methods for control automaton. They are described in [Jac02a]. They could also be proved in the fully-automated style. Jacobi et al. have already presented fully-automated methodology for the verification of fused-multiply-add *FPU*s in [JWPB05]. For verification they used the IBM internal verification tool *SixthSense*.

- We also can try to apply automated methods for all control automatons which we have in the VAMP. Since the state space of all automatons in the VAMP is not large one can try to prove liveness and safety properties by means of model checking. Note that the liveness of the whole processor is still unproved. We have separate proofs of the liveness for all functional units and for the Tomasulo algorithm.

- We also can try to use automated methods for the whole processor in a similar way as it was already done for the verification of a powerful Tomasulo scheduler in [McM98] and as it was described in [BCBZ02, VB99]

- Due to the symmetry we hopefully can do sufficient abstraction for caches. Therefore we could be able to verify caches with the help of model checker as well.

So we conclude that if we want to prove the VAMP automatically we need to invent a good abstract model in order to reduce the state space.

4.3.2 Hardware Optimizations and Extensions

We can do some optimizations and extensions in the hardware in the future. Since the local *MMU* is simple every translated request without exceptions needs to access the memory twice. This is of course slow and does not satisfy the requirements for industrial processors. Therefore we should optimize the local *MMU*.

- All industrial processors with *MMU*s have an internal so called translation look aside buffer. Table look aside buffers (TLB) are used in order to cache page table entries. In this case we do not need to do two requests to the memory if the page table entry which we need was stored in the TLB.

- In the implementation we perform two additions in order to calculate page table entry address. The first one, when we calculate an effective address *dva* and the second one, when we add page index, which is the part of the effective address, with the page table origin. The delay of such computation is obviously equal to twice the delay of the adder. Note that delay of the carry save adder (3/2–adder) is constant [MP00]. Hence we can reduce the delay to delay of one adder and constant delay of 3/2–adder if we compress the sum of these three bitvectors to two bitvectors and after that we compute with a normal adder the page table entry address. This can be done already outside the *MMU*, reducing the latency of *MMU* requests by one cycle. We have now only one reservation station for the memory unit. It makes all proofs easier but implementation of the VAMP is slower. If we increase the number of reservation station we can pipeline the memory unit.

- In our implementation in the case of store accesses we stall execution of this access until the store instruction will be the oldest instruction in the pipeline. Note that for this case it is enough to stall the write request to the memory until this instruction will be the oldest instruction in the pipeline, i.e. the read access to the memory which we do in order to get the page table entry can be started earlier.

- In order to verify computer system which contain hardware we have to extend the hardware model with external devices, i.e. for whole implementation of the virtual memory in hardware we need to have an external device for storage memory pages which are not stored in RAM. Hence, we have to prove a model with swap memory. In [HIP05] the paper and pencil proof of such model is given.

The following extensions in the hardware could be done in the future.

- We also can try to organize multi-level address translation which can be formally specified in the same way as in [Hil05].

- In order to separate memory pages which we can execute, i.e. pages which contain program code and pages which contain data, we can extend the format of the page table entry as follows. We introduce a new flag x (executable). The fetch can be executed only for pages whose flag x is active. Hence, we forbid any access on the instruction fetch to pages which do not contain program code. This extension is also done in many industrial processors in order to distinguish between regions of data and regions of instructions in memory.

4.3.3 Formal Software Verification

The last direction for the future work, which we want to discuss here, is correctness of the software. As a part of Verisoft [Ver03] project we try to verify the following software components.

- Formally verify an compiler of a high level language. Petrova and Leinenbach currently work on verification a compiler for a subset of the C language [Lei05, Pet04]. Apart from some optimization and inline assembler their work is almost finished.

- Formally verify a simple operating system [GHLP05, Bog05].

- Formally verify several applications, e.g. an email client, support of TCP/IP-protocol, and some signature software.

Appendix A

VAMP Instruction Set

The VAMP instruction set is taken from [Bey05] with minimal modifications.

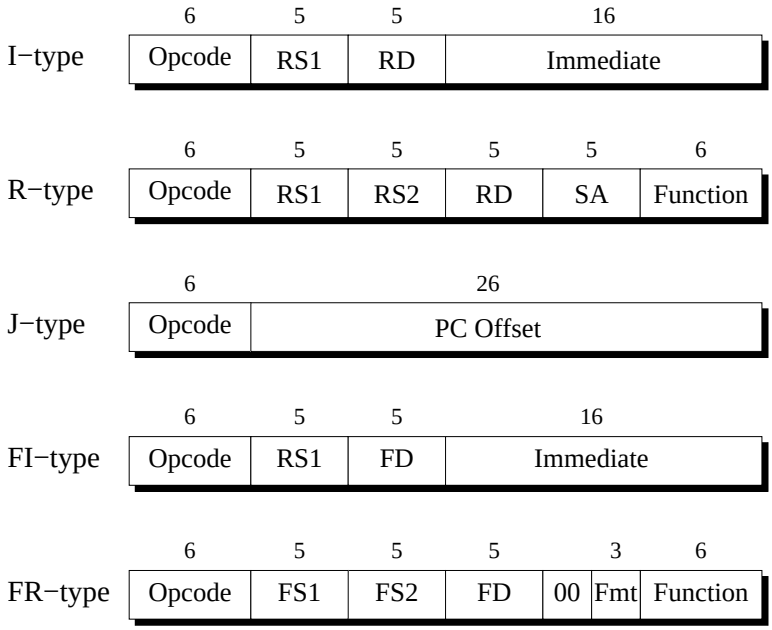

Figure A.1: Instruction Formats of the VAMP

$IR[31:26]$	Mnem.	d	Effect
Memory operations, pa is an effective address or a translated effective address			
100000	`lb`	1	$RD = sext(M[pa + d - 1 : pa])$
100001	`lh`	2	$RD = sext(M[pa + d - 1 : pa])$
100011	`lw`	4	$RD = M[pa + d - 1 : pa]$
100100	`lbu`	1	$RD = 0^{24}M[pa + d - 1 : pa]$
100101	`lhu`	2	$RD = 0^{16}M[pa + d - 1 : pa]$
101000	`sb`	1	$M[pa + d - 1 : pa] = RD[7 : 0]$
101001	`sh`	2	$M[pa + d - 1 : pa] = RD[15 : 0]$
101011	`sw`	4	$M[pa + d - 1 : pa] = RD$
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, pa is an effective address or a translated effective address			
110001	`load.s`	4	$FD[31:0] = M[pa + d - 1 : pa]$
110101	`load.d`	8	$FD[63:0] = M[pa + d - 1 : pa]$
111001	`store.s`	4	$M[pa + d - 1 : pa] = FD[31:0]$
111101	`store.d`	8	$M[pa + d - 1 : pa] = FD[63:0]$
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*.

Name in PVS	Number	Number of commands
processor_inputs_stable_for_request	2.1.8	30
cache_inputs_stable_for_request	2.1.19	30
mmu_correct_hw_dec	2.3.1	40
cache_mr_mw_unary_hw	2.3.2	20
cache_mr_active_until_busy	2.3.3	25
cache_input_stable_hw	2.3.4	575
processor_interface_liveness_hw	2.3.5	615
untranslated_read_hw	2.3.6	610
untranslated_write_hw	2.3.7	815
translated_read_hw	2.3.8	1620
translated_write_hw	2.3.9	1720
addr_is_eq	2.3.10	1005

Table B.1: Lemmas in PVS context `mmu`

Name in PVS	Number	Number of commands
processor_request_is_stable	3.4.3	30
commits_is_req_proc_data	3.4.5	215
sI_inst_helper	3.3.2	40
dmmu_proof	3.4.2	780
vamp_correct_with_interrupt_extended	3.4.1	110
mem_conf_const_helper	3.4.7	15
mem_result_correct_data	3.4.8	1210
memory_unit_out_correct_dpf	3.4.9	540
vamp_induction_step_mem_conf	3.4.10	765
vamp_correct_fetch_PC	3.4.12	40
vamp_induction_step_S1	3.4.13	720
processor_request_is_stable_inst	3.4.14	500
immu_proof	3.4.15	780
terminated_memory_instruction_is_commited	3.4.16	100
mem_conf_helper2	3.4.17	60
self_modifying_helper_ptea	3.4.18	110
self_modifying_helper_pha	3.4.19	70
last_tact_of_req_is_req_proc_inst	3.4.20	180
vamp_correct_without_interrupt	3.5.1	55
vamp_correct_with_interrupt_extended	3.5.5	150
mem_commit_equal_mem	3.5.2	40
vamp_correct_JISR_step	3.5.3	100
vamp_correct_JISR_step_mem	3.5.4	230
vamp_after_last_JISR_is_initial	3.5.6	140
sI_inst_helper	3.5.7	220
sI_inst_correct	3.5.8	300

Table B.2: Lemmas in PVS context `dlxtom_mmu`

Bibliography

[ACHK04] M. Aagaard, V. Ciubotariu, J. Higgins, and F. Khalvati. Combining equivalence verification and completion functions. In *FM-CAD 2004*, volume 3312 of *LNCS*, pages 98–112. Springer, 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.

[BCBZ02] S. Berezin, E. Clarke, A. Biere, and Y. Zhu. Verification of out-of-order processor designs using model checking and a light-weight completion function. In *Formal Methods in System Design*, volume 20, 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.

[Bey05] Sven Beyer. *Putting it all together—Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[BHK94] B. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic Inc., 1994.

[BJ01a] 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.

[BJ01b] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. *Lecture Notes in Computer Science*, 2144:325–328, 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 Hardware Verification'(to appear)*, 2005.

[Bog05] Sebastian Bogan. *Formal Verification of a Simple Operating System (Draft)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[DHP05] I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *CHARME 2005*, LNCS. Springer, 2005.

[GHLP05] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *TPHOLs 2005 (to appear)*, LNCS. Springer, 2005.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss*. PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*. IEEE Computer Society, 2005. To appear.

[HP96a] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HP96b] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[HS99] W. A. Hunt and J. Sawada. Verifying the FM9801 microarchitecture. In *IEEE Micro*, pages 47–55, 1999.

[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.

[IEE85] 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.

[JM98] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[JWPB05] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner. Automatic Formal Verification of Fused-Multiply-Add FPUs. In *DATE*, pages 1298–1303, 2005.

[Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors.* PhD thesis, Saarland University, Saarbrücken, Germany, 2001.

[Lei05] Dirk Leinenbach. *Formal Verification of a Functional Compiler of a C-like Language (Draft).* PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[McM98] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *CAV 98*, volume 1427. Springer, June 1998.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness.* Springer, 2000.

[MS05] Panagiotis Manolios and Sudarshan K. Srinivasan. A parameterized benchmark suite of hard pipelined-machine-verification problems. In *CHARME 2005*, volume 3725 of *LNCS*, pages 363–366. Springer, 2005.

[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.

[Pet04] Elena Petrova. *Formal Verification of Compilers on the Source Code Level (Draft).* PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[SH98] J. Sawada and W. A. Hunt. Processor verification with precise exceptions and speculative execution. In *CAV 98*, volume 1427 of *LNCS*. Springer, 1998.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. In *IEEE Transactions on Computers*, volume C-37, pages 562–573. 1988.

[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.

[Tve05a] Sergey Tverdyshev. Combination of Isabelle/HOL with automatic tools. In *5th International Workshop on Frontiers of Combining Systems (FroCoS)(to appear)*, Lecture Notes in Artificial Intelligence. Springer, 2005.

[Tve05b] Sergey Tverdyshev. *Formal Verification of the VAMP processor by using automated methods (Draft).* PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[VB99] M. Velev and R. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In L. Pierre and T. Kropf, editors, *CHARME 1999*, 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.

[Ver03] The Verisoft Project. http://www.verisoft.de/, 2003.