Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Master's Thesis

The VAMP Memory Unit: Hardware Design and Formal Verification Effort

submitted by

Artem Alekhin

on February 5, 2009

Supervisor
Prof. Dr. Wolfgang J. Paul
Saarland University, Germany

Advisor
Iakov Dalinger, PhD
Saint-Petersburg State University of Civil Aviation, Russia

Rewiewers
Prof. Dr. Wolfgang J. Paul
Saarland University, Germany

Iakov Dalinger, PhD
Saint-Petersburg State University of Civil Aviation, Russia

Contents

List of Figures

Chapter 1

Introduction

Nowadays computer systems seem to be really ubiquitous. Being involved in many significant and even life-critical fields (e.g. nuclear power engineering, the sector of medical technology, automotive engineering, banking, etc.), they are supposed to work in the proper way, whatever happens. Actually the reliability of such systems depends on how accurately the development process of a computer system (both soft- and hardware) is carried out and all design faults are fixed. Even despite the fact that manufacturers spend time and money on testing and debugging, final versions of their products happen to have any kinds of encapsulated errors.

Different bugs can have a wide variety of effects: from slight inconvenience to use a device or a programm to a total crash of a system. Sometimes the results can be extremely serious. So, the European Space Agency's US$1 billion prototype Ariane 5 rocket was destroyed less than a minute after launch in 1996 because of a bug in the on-board guidance computer program. As one more example, the well-known flaw in Intel Pentium floating-point division unit caused a loss of money to replace faulty microprocessors. Unfortunately, the consequences can concern not only money but human health and life. A bug in the code controlling the Therac-25 radiation therapy machine was directly responsible for patient victims in 1996. Apparently, one can find more examples.

Such incidents take place because of the so-called human factor and testing limitations. The former means that the developers do not take into account some details of a system behaviour or just make mistakes. The latter is a bottleneck caused by an increasing amount of input data and working scenarios for more complex hard- and software to be tested on currently used tools. In many cases application of only non-exhaustive tests is possible.

The only way to avoid such failures is to cover all possible cases of system work. This turns to be feasible with *formal verification*, the technique of proving or disproving the correctness of a system with respect to its formal specification. The proofs are made on abstract mathematical models and involve either *logical inference* or exploring (*model checking*) all their states and transitions in reduced computing time. In this approach guaranteeing the absence of human errors and gaps in models is usually achieved by applying special proof assistant tools. However, in spite of many scientists' efforts, the ideal methods of formal verification are left to be found.

Verisoft [Ver03] is a long-term research project, which aims at the pervasive formal verification of computer systems. A part of the Verisoft project is an entire academic system covering all general-purpose computer system layers: the gate-level hardware, system software, networking and communication, applications, and a compiler. The hardware platform is called Verified Architecture Microprocessor (VAMP) [BJK$^+$03].

This master thesis is devoted to the formal verification of the VAMP memory unit (*MU*) and based on the work carried out in [Dal06]. The new design of the *MU*, developed here, contains translation look-aside buffers (*TLB*) for fast virtual address translation inside the memory management units (*MMU*) and supports accesses to external devices.

A computer-aided verification tool used throughout the whole work is an interactive theorem prover Isabelle/HOL[1] bound [Tve05] with the NuSMV [CCG$^+$02] and SMV [McM99] model checkers. The results (correctness proofs and models of hardware blocks) are presented as Isabelle mathematical theories. The work is described formally and paper-and-pencil proofs are provided.

Outline

This thesis is organized in 5 chapters.

- In the rest of Chapter 1 we sketch the verification technique used in this work. Besides, we give a short description of the VAMP.
- In Chapter 2 we introduce the notation, gates and basic circuits used in this work. We also cover the foundations of the virtual memory mechanism.
- In Chapter 3 we present a new construction of the memory management unit that is a hardware supporting virtual memory. We also prove its correctness with respect to assumptions for the processor, *TLB*, and memory system.
- In Chapter 4 we describe the VAMP memory unit with the new *MMU*s (Chapter 3) and an interface for external devices. Providing the *MU* specification in the VAMP and correctness proofs we do not focus on operations with external devices that are well considered in [Tve08].

[1]http://isabelle.in.tum.de

- In Chapter 5 we summarize the results and give directions for future work.

1.1 Verification Technique

The formal verification of the memory unit is naturally done with respect to *proof decomposition*. This principle reasonably reflects the situation in the real world: huge components are divided into relatively small and independent pieces. Having been implemented and verified, such blocks can be repeatedly utilized without further exploration of their behaviour.

For units that provide results of computations in one iteration (some adders, shifters, logic circuits, etc.), modelling of their implementation and specifying result properties are considered to be trivial with units' complete design.

As for hardware (like *MMU*s, whole *MU*) that has internal configuration and requires one or more cycles to complete an operation, the implementation is modelled with *step functions*:

$$\delta(c, in) = (c', out),$$

where c, $c' \in C$ - the current and the next configurations of a unit, $in \in In$ and $out \in Out$ - its current input and output signals. A hardware configuration is a content of registers and addressable RAMs that might be in the unit's construction. Particularly, the RAM configuration is considered as an uninterpreted function mathematically updated in case of writing at a given position.

The specification for such hardware is rather complicated. So, for the memory management units it is based on signal sequences in time and memory updates. Furthermore, for the whole processor and the memory unit as its component the specification corresponds to the programmer's model. In this model a programmer works with visible registers and memory. The processor executes an instruction in each computation step. Hence, it is specified with a step function based on the visible machine configuration, external inputs, and specification computations for each instruction to be executed.

Since the processor blocks covered in this work are detailed and fixed in Isabelle/HOL theories, with translation tools they can be used to generate real hardware. However, in the thesis for some memory unit components used before in [Dal06] we do not provide their design description and correctness proofs.

The correspondence between the specification and implementation for the memory unit and its parts is presented in a set of lemmas proven in Isabelle/HOL and accompanied with paper-and-pencil proofs. In both cases (except as model checking is involved) we use the same standard techniques as primitive induction, natural deduction, rewriting and simplification, case splitting, etc. This makes pencil-and-paper proofs conform with those driven by the interactive theorem prover. Note that only crucial lemmata are covered in the paper. The reason is that a lot of claims are clear for human reasoning and their formal proofs are only needed for complete verification effort in Isabelle.

1.2 The VAMP in Brief

The VAMP is a pipelined 32-bit RISC processor with DLX-instruction set [HP96, MP00] including IEEE-compliant floating point instructions.

It features a delayed PC with one delay slot, an out-of-order Tomasulo scheduler exploiting a reorder buffer [Krö01], and provides support for virtual memory [Hil05, Dal06, DHP05] and maskable precise interrupts.

The VAMP has five execution units: a fixed point unit (XPU), three floating point units (FPU) [Ber01, BJ01, Jac02a, Jac02b], and the memory unit.

The memory interface of the VAMP contains two *MMU*s that access instruction and data caches [Bey05] respectively, as well as external devices for data word retrieving and writing. The caches implementing a write-back algorithm, in turn, work with the main memory via a bus protocol [MP00, Bey05]. As device data accesses should not be cached, we provide a special logic in the *MU* for direct accesses to devices via the bus protocol.

Chapter 2

Basics

This chapter begins with introducing a notation and definitions used in the whole thesis. Then it covers virtual memory foundations underlying the VAMP virtual memory support. At the end of the chapter the description of gates and some basic circuits used in the *MU* construction is provided.

2.1 Notation

The notation and some definitions presented here are borrowed from Dalingers's work [Dal06].

For the whole thesis $\mathbb{N}$ defines the set of natural numbers *including 0* and $\mathbb{N}^+ := \mathbb{N}\backslash\{0\}$. The set of integer numbers is denoted by $\mathbb{Z}$. For the set of real numbers we use $\mathbb{R}$. We start with a shorthand notation for subsets of $\mathbb{Z}$.

Definition 2.1.1 *Let n, $m \in \mathbb{Z}$ be integer numbers. We define the following **integer intervals**:*

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

To argue about signals and computations in the processor we proceed with defining standard notions such as words and vectors of bits.

Definition 2.1.2 *Let $\Sigma \neq \emptyset$ be a set called* ***alphabet****. A* ***word*** *of length $n \in \mathbb{N}$ over the alphabet Σ is a function $w : \mathbb{Z}_n \to \Sigma$. A word w is uniquely identified by the n-tuple of values $(w(n-1), w(n-2), \ldots,\ w(0))$. For a laconic style we also use $w_{n-1}w_{n-2}\ldots w_0$ or just $w[n-1:0]$. The set of all words of length n over Σ we define as $\Sigma^n := \{w | w : \mathbb{Z}_n \to \Sigma\}$.*

Definition 2.1.3 *For a nonempty word w of length $n \in \mathbb{N}^+$ and natural numbers $k \in \mathbb{Z}_n$, $l \in \mathbb{Z}_{\leq k}$ a word part $w[k:l]$ is a* ***subword****. For the case $w[k:k]$ we just use a shorthand notation $w[k]$ denoting* ***k-th element*** *of the word w.*

Definition 2.1.4 *The* ***concatenation*** *of words $a \in \Sigma^n$, $b \in \Sigma^m$ for any $n, m \in \mathbb{N}^+$ is defined as follows*

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

Instead of writing the infix operator $\circ$, we simply denote concatenation by $a[n-1:0]b[m-1:0]$.

Definition 2.1.5 *We call an alphabet also a* ***domain****. In the scope of the thesis we denote a domain consisting of bits by $\mathbb{B} := \{0, 1\}$.*

Definition 2.1.6 *A* ***bitvector*** *of length n is a word of length n over the domain $\mathbb{B}$. A subword of a bitvector is a* ***subbitvector****. Analogously, for a bitvector v of length $n \in \mathbb{N}^+$ and a number $k \in \mathbb{Z}_n$ we call its k-th element also the* ***k-th bit*** *of the bitvector v.*

Note that to avoid conversions between bit and boolean values we associate the bit values *1* and *0* with the values *true* and *false* of boolean type correspondingly. This allows us to write more understandable predicates and formulae. However, in Isabelle theories we use bitvectors without such a trick because its standard libraries contain a plenty of already proven lemmas on this type.

Definition 2.1.7 *For any $a \in \mathbb{B}^n$ we define a function $twice_n : \mathbb{B}^n \to \mathbb{B}^{2\cdot n}$ of* ***duplicating*** *the bitvector:*

$$twice_n(a) := a \circ a$$

If the length of a bitvector is devisable by 8 we can refer to its any 8 successive bits as a byte. For this purpose we introduce the next shorthand notation:

Definition 2.1.8 *For any $B \in \mathbb{N}^+$, $b \in \mathbb{N}$, such that $b < B$, and a bitvector $w \in \mathbb{B}^{8\cdot B}$, the selection of the* ***b-th byte*** *of the bitvector w is denoted as*

$$|w|_b := w[8 \cdot b + 7 : 8 \cdot b]$$

Definition 2.1.9 *For any $B, d \in \mathbb{N}^+$, $b \in \mathbb{N}$, such that $b < B$, $d \leq B$, $b + d \leq B$ and a bitvector $w \in \mathbb{B}^{8 \cdot B}$, the selection of* ***d adjacent bytes*** *of the bitvector w is denoted as*

$$|w|_{b,d} := |w|_{b+d-1} \cdots |w|_b$$

We proceed with the standard interpretation of binary numbers [MP00].

Definition 2.1.10 *Let $n \in \mathbb{N}^+$ and $a \in \mathbb{B}^n$. We denote by*

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

a natural number with a ***binary representation*** *a. In this case the bitvector a is called a* ***binary number****. Note that the function $\langle \cdot \rangle : \mathbb{B}^n \to \mathbb{Z}_{2^n}$ is bijective.*

Definition 2.1.11 *A function*

$$\text{bin}_n := \langle \cdot \rangle^{-1}, \ \text{bin}_n : \mathbb{Z}_{2^n} \to \mathbb{B}^n$$

returns the ***n-bit binary representation*** *of a natural number.*

For the proofs we will need a few statements concerning the bitvectors.

Proposition 2.1.12 *For $n \in \mathbb{N}^+$ and $a \in \mathbb{B}^n$ the following holds:*

$$\left\langle a \circ 0^2 \right\rangle \bmod 8 = 0 \Longleftrightarrow \neg a[0]$$

Proposition 2.1.13 *For $n \in \mathbb{Z}_{\geq 2}$ and $a \in \mathbb{B}^n$ the following holds:*

$$\text{bin}_{n-1}(\left\lfloor \left\langle a \circ 0^2 \right\rangle / 8 \right\rfloor) = a[n-1:1],$$

where for $k \in \mathbb{R}$ the notation $\lfloor k \rfloor$ is rounding down to the nearest integer.

Both the propositions are proven with the help of properties already prepared in Isabelle/HOL.

Among bitvector operations we introduce addition returning a bounded result.

Definition 2.1.14 *For $a, b \in \mathbb{B}^n$ and $n \in \mathbb{N}^+$ a function $+_n : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$ denotes* ***n-bit binary addition*** *and is defined as follows:*

$$a +_n b := \text{bin}_n(\langle a \rangle + \langle b \rangle \bmod 2^n)$$

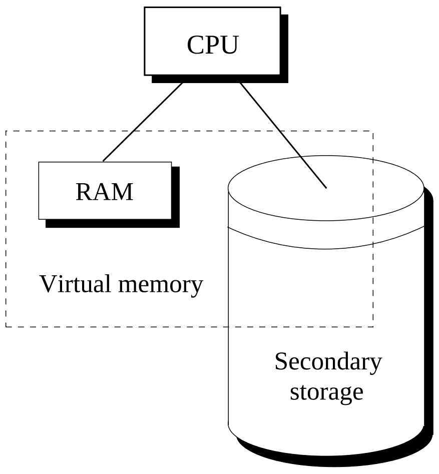

Figure 2.1: Organization of Virtual Memory

2.2 Virtual Memory

Many software applications require larger physical memory than it is installed in the computer system. This takes on special significance when the operating system supports multiple processes running seemingly in parallel. The well known solution of this problem is exploiting *virtual memory*, a memory management technique, used by multitasking computer operating systems.

2.2.1 Virtual Memory Conception

Virtual memory (see Figure 2.1) was developed to automate the movement of a programm code and data between the main memory and any secondary storage to give the appearance of a single large store [JM98]. So, including both the levels of the memory hierarchy, such a management extends a process's address space.

A well designed virtual memory system keeps only the most often used portions of a process's address space in the main memory whereas the left burden is stored on a disk (so called *swap memory*) and retrieved as needed. This is provided by the operating system and hardware found in the memory unit with the *MMU*s supporting translating *virtual addresses* in the process's address space to their *physical* equivalents for the main memory. The whole mechanism acts in a way that is invisible to the rest of the software running on the computer.

Usually virtual memory allows to simulate the main memory of any size limited only by the address width. So, for a 32-bit operating system, the maximal size of the virtual memory is 2^{32}, or 4 gigabytes. This value can be substantially increased in case of newer 64-bit computer systems with 64- or

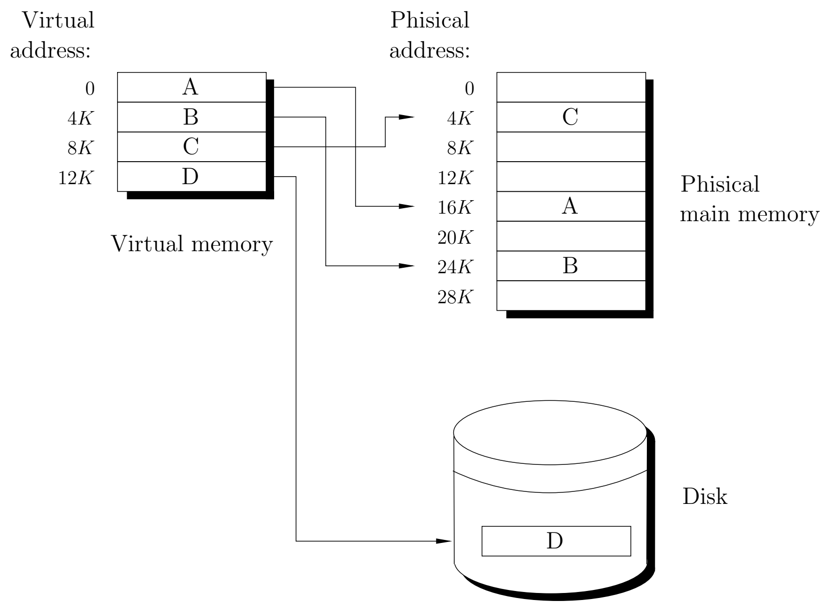

Figure 2.2: Example of Virtual Memory Allocation

48-bit addressable memory space.

Managing the memory hierarchy described above allows to share a small amount of the physical memory among processes running at any instant in time. Memory blocks are allocated to different processes that should be restricted to accessing only their own portions of memory. The only approach to achieve this is using some kind of a *protection* scheme.

In addition to the memory hierarchy management and sharing address space, virtual memory also simplifies loading programmes for execution [HP96]. With *relocation* it is possible to run a programm in any location in the main memory. The programm can be placed anywhere in the main memory or on the secondary storage by just changing the address mapping (e.g., Figure 2.2).

As one more point of using the virtual memory mechanism, one can figure out reduction of the time for starting a programm. The reason is that not the whole programm code and data are needed for that.

So, virtual memory makes the job of an application programmer simpler. However, frequent requests to the hard disk in the virtual memory mechanism greatly reduce the performance of applications. Therefore, the programmer should take it into account and optimize the code to work with memory in a better way.

2.2.2 Implementation Details

The implementation of the virtual memory mechanism depends on a processor architecture and involves different manufacturers' features. Covering industrial processors' design is out of the scope of this thesis. Below, we concentrate only on common principles, and point out the details underlying the virtual memory support used in the VAMP.

Virtual memory systems can be categorized in two classes: with fixed-size blocks, called *pages*, and variable-size *segments*. The first one is easier to be maintained. Based on [Dal06] we proceed using *paging* with the page size of 4 Kilobytes. Note, typically the page size is in the range of 4096 to 65536 bytes.

In paging, the low order bits of the binary representation of the virtual address are considered as an offset within the page and used as the low order bits (*byte index*) of the translated physical address. So, for the page size of 2^n bytes, the byte index consists of n least significant bits of the virtual address.

The high order bits, called *virtual page index*, are treated as a key to one or more address translation tables (*page tables*) providing the mapping to the physical memory addresses of the pages (*physical page indices*). The physical page index retrieved is used as high order bits of the actual physical address.

The page tables are maintained by the operating system and are stored in the physical memory. However, in other industrial implementations they are usually swapped because of their substantial size. The page table can be organized in different ways [JM98]. In this work, we are bounded by the simplest one: the table is just an array of *page table entries*, in short, *pte*. Each *pte* contains the physical page index and some additional information, such as:

- whether the page contains data or executable code,
- whether the data is protected and only for the system use,
- whether the page is stored in the physical memory or on the hard disk.

Since each page table is identified with a *page table origin* in the main memory, the address of a *pte* is computed as an offset (given by the virtual page index) with regard to this starting position.

For the same running application the pages may be both in the physical memory and on the hard disk in the swap area. If during a processor request a page is not resident in the main memory, it must be copied from the disk, or, in other words, *paged in.* This is known as *demand paging.* When the memory space is needed for that, a page replacement policy [ASG04] is applied to choose a particular page and write it back to the disk, so that the space is freed up for more active pages.

In the situation when a page is inaccessible in the main memory, an exception must be raised in the processor. The *MMU* generates a so called *page fault* to signal about the problem (more detail in Chapter 3, Section 3.1.2). As a result, a special handler is called. If the page resides in the swap memory, the handler invokes a swap operation to bring the page to the physical memory. The details on how the page swap operation is performed are out of the scope of this thesis. The algorithm and implementation depends on the particular operating system.

2.2.3 Fast Address Translation

As one could notice in the implementation described above, to retrieve data or an instruction from the memory the *MMU* must produce the memory request twice. Following this scenario is costly indeed because of a substantial latency in the memory.

A remedy for that is in minimizing the performance penalty of the address translation. One way is to remember the last translation, so that the mapping process is skipped if the request refers to the same page as the last one. A more general way is based on exploiting the temporal locality [MP00] for page table entries. Having been accessed, they are stored in a distinct cache called a *translation look-aside buffer* (*TLB*).

The *TLB* entry is analogous to the cache one with the tag holding a part of the *pte* address and the *pte* as data. When the processor generates a request with a virtual memory address, the hardware searches the *TLB* for the appropriate address mapping. If the translation exists, the request can be continued without accessing the page table. Otherwise, the *TLB* must be filled with the correct mapping information.

Refilling on the *TLB* miss is performed either by the operating system or by the hardware. With a hardware-managed *TLB*, the hardware is responsible for generating an additional memory request to read the page table and provide the information for the *TLB*. For the software-managed *TLB*, an interrupt mechanism invokes a *TLB* miss handler to carry out refilling.

The drawbacks of the latter scheme are connected with the interrupt handling. If the handle code is not in the instruction cache, this causes an additional delay. Besides, there is a need to flush the pipeline. Obviously, removing many instructions from the reorder buffer affects the system performance.

In this work the instruction and data *MMU*s are accompanied by the hardware-managed *TLB*s. Such a choice is dictated not only by the reasons covered above. The proof of the *MMU* local correctness is less complicated this way. Otherwise, we would require the correctness of the operation system software handling *TLB*'s misses.

The *TLB* is based on the direct mapped cache [Bey05] and refilled under control of the *MMU* automation. Since the *TLB* acts like a buffer and does

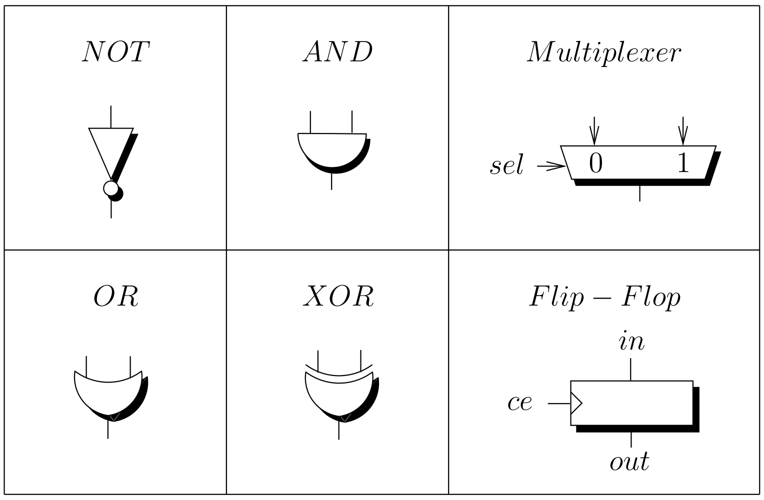

Figure 2.3: Symbols of the Basic Gates

not support write-back algorithm, it is cleared when the system rewrites the page table. The proof of the *TLB*'s correct behaviour is out of the scope of this work.

2.3 Gates

The basic gates composing all the circuits constructed in this work are depicted on Figure 2.3. With the standard notation we can represent them as functions and specify their results.

For inputs $a, b \in \mathbb{B}$, the gates AND, OR, XOR and the inverter (NOT):

- $AND : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, $AND(a, b) = a \wedge b$
- $OR : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, $OR(a, b) = a \vee b$
- $NOT : \mathbb{B} \rightarrow \mathbb{B}$, $NOT(a) = \neg a$
- $XOR : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, $XOR(a, b) = a \oplus b = \neg a \wedge b \vee a \wedge \neg b$

With an additional input $sel \in \mathbb{B}$, the multiplexer (MUX) is

- $MUX : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$,
  $$MUX(a, b, sel) = \begin{cases} a & \text{if } sel = 1 \\ b & \text{otherwise} \end{cases}$$

For input and output signals $in, out \in \mathbb{B}$, a clock enable $ce \in \mathbb{B}$, current and the next configurations $c, c' \in \mathbb{B}$, the flip-flop can be described by a step function:

- $FlipFlop : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \times \mathbb{B}$,
  $FlipFlop(in, c, ce) = (out, c')$,
  where $out = c$, and $c' = \begin{cases} in & \text{if } ce = 1 \\ c & \text{otherwise} \end{cases}$

Since an *n-bit register* consists of n flip-flops, it can be depicted and described in the same way except its capacity.

Note, for a more concise description, in spite of exploiting the gate names in formulae, we shall use only the corresponding logical connectives, i.e. $\neg$, $\wedge$, $\vee$, and $\oplus$. As for the multiplexer, it is naturally substituted by the "if clause".

2.4 Basic circuits

In this section we provide definitions for basic circuits used in the *MU* construction. Since their implementation details and correctness proofs were covered in [Krö01, Dal06], we do not go into details. Now all the circuits are realized and verified in Isabelle/HOL.

Definition 2.4.1 *For $n \in \mathbb{N}^+$, inputs $a, b \in \mathbb{B}^n$, and $c \in \mathbb{B}$, an* ***n-bit adder*** *is a circuit defined as*

$$\begin{aligned} &Add_n : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \rightarrow \mathbb{B}^{n+1}, \text{ such that} \\ &Add_n(a, b, c) = \text{bin}_{n+1}(\langle a \rangle + \langle b \rangle + \langle c \rangle). \end{aligned}$$

Note that we use the *carry lookahead adder* [MP00] because of its low delay.

Definition 2.4.2 *For $n \in \mathbb{N}^+$ and inputs $a, b \in \mathbb{B}^n$, an* ***n-bit less tester*** *is a circuit defined as*

$$\begin{aligned} &Less_n : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}, \text{ such that} \\ &Less_n(a, b) = (\langle a \rangle < \langle b \rangle). \end{aligned}$$

This circuit is just a part of the *n-bit adder and subtractor*, given in [Dal06].

Definition 2.4.3 *For $n \in \mathbb{N}^+$ and an input bitvector $a \in \mathbb{B}^n$, an* ***n-bit AND-tree*** *is a circuit defined as*

$$\begin{aligned} &AND_n : \mathbb{B}^n \rightarrow \mathbb{B}, \text{ such that} \\ &AND_n(a) = \bigwedge_{i=0}^{n-1} a[i]. \end{aligned}$$

Such a tree for a particular size of the input bitvector is easily verified in Isabelle/HOL with the help of NuSMV.

Chapter 3

Memory Management Unit with TLB

The work presented in this chapter is based on Dalinger's work [Dal06] and realizes the *MMU* extensions proposed in [Hil05, Dal06]. We introduce the specification and the implementation of the memory management unit with the translation look-aside buffer and show its local correctness, i.e. without embedding the module into the processor. All the proofs are carried out in Isabelle/HOL and provided in the pencil-and-paper form here.

3.1 Specification of the MMU

3.1.1 MMU Specification Configuration

Since the *MMU* communicates with the processor, the memory, and uses the *TLB*, we specify corresponding interfaces (Figure 3.1). A set of all signals in an interface is called an $interface\ observation$.

Note, that we do not ascribe an additional control signal *purge* for *TLB* to one of three interfaces of the memory management unit. However, it is needed for covering the *MMU* specification.

Definition 3.1.1 *We define a set of MMU specification configurations as*

$$C_{spec} := Iobs_p \times Iobs_t \times Iobs_m \times \mathbb{B} \times Memory,$$

where

- $Iobs_p$ *– processor interface observations,*
- $Iobs_t$ *– TLB interface observations,*
- $Iobs_m$ *– memory interface observations,*
- $Memory$ *– configurations of the physical memory.*

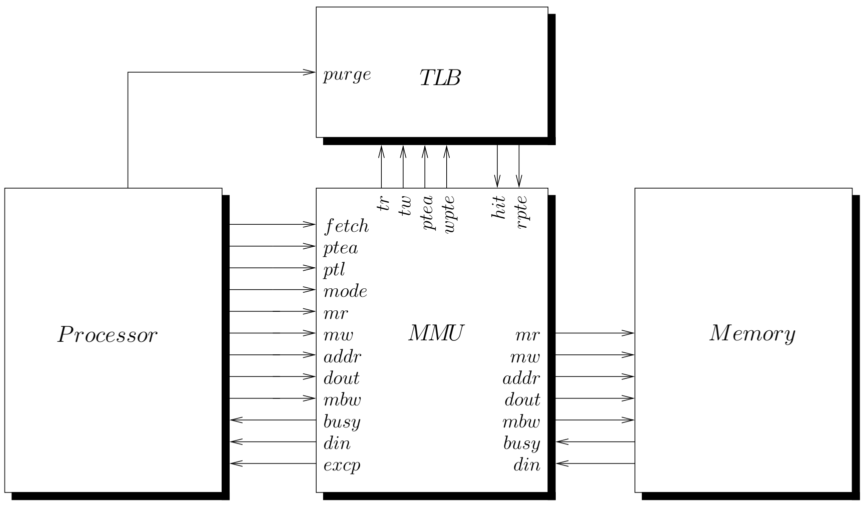

Figure 3.1: Interfaces of *MMU* with *TLB*

An element $c \in C_{spec}$ is a tuple:

$$\begin{aligned} c \;:=\; & (iobs_p \in Iobs_p, iobs_t \in Iobs_t, iobs_m \in Iobs_m, \\ & purge \in \mathbb{B}, mem \in Memory) \end{aligned}$$

The processor interface observation $iobs_p$ is a 13-tuple:

$$\begin{aligned} iobs_p \;:=\; & (mode, mr, mw, fetch, mbw, addr, \\ & ptea, ptl, dout, reset, din, busy, excp) \end{aligned}$$

The following components of the tuple are inputs for the *MMU* from the processor side:

- $mode \in \mathbb{B}$ – a memory access type. If this flag is set, the processor performs a translated memory access and runs in *user mode*, otherwise the processor is in *kernel mode* and an untranslated memory access is in progress.
- $mr \in \mathbb{B}$ – a flag indicating a memory read operation.
- $mw \in \mathbb{B}$ – a flag indicating a memory write operation.
- $fetch \in \mathbb{B}$ – a flag indicating an instruction fetch access.
- $mbw \in \mathbb{B}^8$ – memory byte write signals. As each memory request accesses a double word, the signals *mbw* indicate which bytes are to be written. This component is used only in case of a write access.
- $addr \in \mathbb{B}^{29}$ – a memory access address. Depending on the flag *mode*, the address is either a virtual or a physical one.

- $ptea \in \mathbb{B}^{30}$ – a page table entry address. This address is used to access an entry of the page table.
- $ptl \in \mathbb{B}^{20}$ – the page table length. It is used only for translated memory accesses and shows the size of the page table.
- $dout \in \mathbb{B}^{64}$ – data to be stored in the memory. This data is used for write accesses only.
- $reset \in \mathbb{B}$ – a flag of the processor reset.

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

- $din \in \mathbb{B}^{64}$ – data read from the memory. The data output is used for read accesses only.
- $busy \in \mathbb{B}$ – a *busy* flag. It signals that the *MMU* perform an operation and is busy to the processor.
- $excp \in \mathbb{B}$ – an exception flag. This flag is set, if the memory operation is terminated abnormally without updating the memory or providing data retrieved.

The *TLB* interface observation is a 6-tuple:

$$iobs_t := (tr, tw, ptea, wpte, hit, rpte)$$

The inputs for the *TLB* are:

- $tr \in \mathbb{B}$ – a flag indicating a read access to the *TLB*.
- $tw \in \mathbb{B}$ – a flag indicating a write access to the *TLB*.
- $ptea \in \mathbb{B}^{30}$ – a page table entry address for reading/writing a page table entry in the *TLB*. It is equal to the *ptea* from the processor.
- $wpte \in \mathbb{B}^{32}$ – a page table entry to be written into the *TLB*.

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

- $hit \in \mathbb{B}$ – a *hit* flag. It is set, if the *TLB* contains a page table entry addressed with *ptea*.
- $rpte \in \mathbb{B}^{32}$ – a page table entry read from the *TLB*. This output presents appropriate data only if the signal *hit* is raised up.

The auxiliary input signal *purge* coming directly from the processor causes purging the *TLB* when the content of the *TLB* happens to be inconsistent with the page table.

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

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

The inputs to the memory are:

- $mr \in \mathbb{B}$ – a flag indicating a memory read access.
- $mw \in \mathbb{B}$ – a flag indicating a memory write access.
- $mbw \in \mathbb{B}^8$ – memory byte write signals equal to *mbw* supplied by the processor.
- $addr \in \mathbb{B}^{29}$ – an address for a physical memory location.
- $dout \in \mathbb{B}^{64}$ – data to be written into the memory.

The rest of the components are outputs from the memory:

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

We use a record notation for the defined interfaces as well as for the specification configuration in whole, e.g. the component *din* of $iobs_m$ is denoted by $iobs_m.din$.

Definition 3.1.2 *Depending on the read, write and fetch signals we define a new signal for the interface* $iobs_p$ *by*

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

and similarly for the interface $iobs_m$ *by*

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

Note also that we use a short notation $iobs_p.inputs$ *(*$iobs_m.inputs$*) for the MMU (physical memory) inputs from the processor (MMU) respectively. We also use similar abbreviations* $iobs_p.outputs$ *(*$iobs_m.outputs$*) for the MMU (memory).*

Definition 3.1.3 *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:*

$$mem \in Memory := \{f : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}\}$$

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

Definition 3.1.4 *We define the set of all specification traces as follows:*

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

For the further description and explanation of the *MMU* with the *TLB* we will use a more convenient notation, namely:

Definition 3.1.5 *Let s be any signal from the interfaces $iobs_p$, $iobs_m$ or $iobs_t$. We introduce a short notation s_x^t for $trc(t).iobs_x.s$ where $t \in \mathbb{N}$ denotes a hardware cycle, x denotes one of three interfaces (p, m, t for the processor, memory and TLB interfaces correspondingly) and $trc \in Trace$, e.g. $trc(t).iobs_p.busy$ and $busy_p^t$ are the same for us.*

Definition 3.1.6 *We use a short notation $purge^t$ for the signal $trc(t).purge$ where $t \in \mathbb{N}$ denotes a hardware cycle, $trc \in Trace$, and $purge \in \mathbb{B}$.*

Analogously for the memory one defines the similar shorthand notation.

Definition 3.1.7 *We use a short notation mem^t for $trc(t).mem$ where $t \in \mathbb{N}$ denotes a hardware cycle, $trc \in Trace$, and $mem \in Memory$.*

3.1.2 MMU Operations

The *MMU* realizes 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. If the translation of the virtual address fails, the memory operation invoked by the processor is not performed. In such a case the *MMU* should signal an exception to the processor.

These operations will be precisely specified in the section 3.1.4. Now we only consider the address translation mechanism used in the *MMU*.

For the translated operations we define an additional function similar to the one described in [Dal06]. This is the *decode implementation translation* function, or $DecodeITr$:

$$DecodeITr : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{20} \times \mathbb{B}^{30} \times \mathbb{B}^{32} \times Memory \rightarrow \mathbb{B} \times \mathbb{B}^{32} \tag{3.1}$$

Figure 3.2 shows the principle of the address translation done by $DecodeITr$ function.

This function takes the following inputs

- mr, mw, $fetch \in \mathbb{B}$ – a type of a memory access: read, write, fetch,
- $ptl \in \mathbb{B}^{20}$ – a page table length,
- $ptea \in \mathbb{B}^{30}$ – a page table entry address,

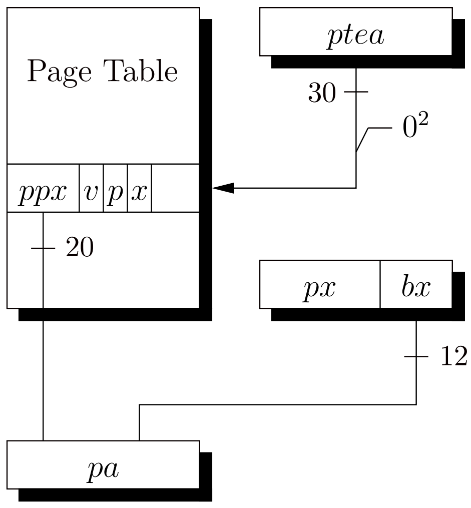

Figure 3.2: Address Translation for Virtual Address

- $va \in \mathbb{B}^{32}$ – a virtual address,
- $mem \in Memory$ – a memory configuration.

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 a physical address. For the calculation of *excp* and *pa* we define a few intermediate values. The virtual address is decomposed into the page index $px \in \mathbb{B}^{20}$ and the byte index $bx \in \mathbb{B}^{12}$:

$$va = px \circ bx \tag{3.2}$$

Each page table entry *pte* is 4 bytes wide. The number of page table entries in the page table is equal to $ptl + 1$. The page table entry address $ptea \in \mathbb{B}^{30}$ is precomputed in the processor using *px* and the address of the page table origin.

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

Let

$$pteaddr = \text{bin}_{29}(\lfloor \langle ptea \circ 0^2 \rangle /8 \rfloor)$$

in

$$pte := \begin{cases} mem(pteaddr)[31:0] & \text{if } \langle ptea \circ 0^2 \rangle \bmod 8 = 0 \\ mem(pteaddr)[63:32] & \text{otherwise} \end{cases} \tag{3.3}$$

The page table entry $pte \in \mathbb{B}^{32}$ (see Figure 3.3) consists of several fields:

- $ppx := pte[31:12]$ – a physical page index,

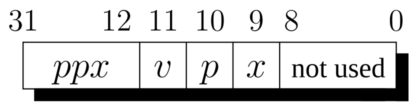

Figure 3.3: Page Table Entry

- $v := pte[11]$ – a valid bit indicating whether a page is in the physical memory or on the secondary storage,
- $p := pte[10]$ – a write protection bit,
- $x := pte[9]$ – an execution bit,
- $pte[8:0]$ – the last nine bits are not used.

During an address translation operation an exception happens in the following cases:

- the page index is larger then the page-table length, i.e. such an access would be outside the page table,
- there is a write access and a page is protected,
- there is an instruction fetch but a page is not executable,
- a page is not in the physical memory, as indicated by the valid bit.

We can formulate *excp* in the following equations:

$$\begin{aligned} lexcp &:= \langle px\rangle > \langle ptl\rangle && (3.4)\\ pteexcp &:= (mw \wedge p) \vee (fetch \wedge \neg x) \vee \\ &\quad\ ((mr \vee mw \vee fetch) \wedge \neg v) && (3.5)\\ excp &:= lexcp \vee pteexcp && (3.6)\end{aligned}$$

The *physical address* is calculated so that in case of $\neg excp$ it is a concatenation of the physical page index and the byte index. Otherwise, pa is set to 0^{32}:

$$pa := \begin{cases} 0^{32} & \text{if } excp \\ ppx \circ bx & \text{otherwise} \end{cases} \qquad (3.7)$$

Using the variable defined above we show the result of a translation *tr* of a processor request starting in a cycle *t* as follows:

$$
\begin{aligned}
&\textit{Let}\\
&\qquad dec = DecodeITr(mr_p^t, mw_p^t, fetch_p^t, ptl_p^t, ptea_p^t, addr_p^t \circ 0^3, mem^t)\\
&\textit{in}\\
&\qquad tr(t).pa := dec.pa,\\
&\qquad tr(t).excp := dec.excp,\\
&\qquad tr(t).lexcp := lexcp,\\
&\qquad tr(t).pteexcp := pteexcp,\\
&\qquad tr(t).pte := pte,\\
&\qquad tr(t).ppx := ppx,\\
&\qquad tr(t).bx := bx.
\end{aligned}
\tag{3.8}
$$

These results will be used for the further specification and correctness proofs of the *MMU*.

3.1.3 Assumptions for the MMU

In this section we make assumptions on the input signals of the *MMU*. For the further work we will need to prove these assumptions while integrating the *MMU* into the processor.

First, we define properties for the processor interface. The properties should be written as predicates on traces. For the whole section $trc \in Trace$ denotes a trace.

Definition 3.1.8 *We call the signals read, write and fetch of the processor interface mutually exclusive if the following predicate holds:*

$$
\begin{aligned}
proc\_mr\_mw\_fetch\_mutexc(trc) :=&\\
\forall t \in \mathbb{N} : \neg(mr_p^t \wedge mw_p^t) \wedge&\\
\neg(mr_p^t \wedge fetch_p^t) \wedge&\\
\neg(mw_p^t \wedge fetch_p^t)&
\end{aligned}
$$

Definition 3.1.9 *We call the input signals of the processor interface stable if the following predicate holds:*

$$
\begin{aligned}
proc\_inputs\_stable(trc) :=&\\
\forall t \in \mathbb{N} : req_p^t \wedge busy_p^t \implies&\\
inputs_p^t = inputs_p^{t+1}&
\end{aligned}
$$

Definition 3.1.10 *We define the predicate is_req_proc for a processor request in the following way:*

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

Finally the predicate of the whole processor interface correctness criteria is defined as follows:

Definition 3.1.11 *The MMU inputs from the processor are called correct if they are stable and the read, write and fetch signals are mutually exclusive. Formally:*

$$\begin{aligned}
&good\_proc\_interface(trc) := \\
&\qquad proc\_mr\_mw\_fetch\_mutexc(trc) \wedge \\
&\qquad proc\_inputs\_stable(trc)
\end{aligned}$$

For the interface between the *MMU* and the *TLB* we make an assumption that guarantees in case of a hit the *TLB* provides consistent data: a page table entry currently residing in the page table in the memory.

Definition 3.1.12 *We call the TLB data consistent if the following predicate holds:*

$$\begin{aligned}
&tlb\_data\_consist(trc) := \\
&hit_t^t \implies \exists t' \in \mathbb{Z}_t : tw_t^{t'} \wedge \\
&\qquad\qquad rpte_t^t = wpte_t^{t'} \wedge \\
&\qquad\qquad ptea_t^t = ptea_t^{t'} \wedge \\
&\qquad\qquad \forall t'' : (t'' \in \big[t' : t\big[\implies \neg purge_t^{t''}) \wedge \\
&\qquad\qquad\qquad (t'' \in \big]t' : t\big[\implies \neg(tw_t^{t''} \wedge ptea_t^{t''} = ptea_t^{t'}))
\end{aligned}$$

Beside the interface between the *TLB* and the *MMU*, we provide a description for the signal *purge*. Since the page table origin, *ptl*, or the page table can be modified in the kernel mode, we connect *purge* with the corresponding signal.

Definition 3.1.13 *We define the predicate tlb_purge_comp for the TLB in the following way:*

$$\begin{aligned}
&tlb\_purge\_comp(trc) := \\
&\qquad \forall \in \mathbb{N} : \ \neg mode_p^t \implies purge^t
\end{aligned}$$

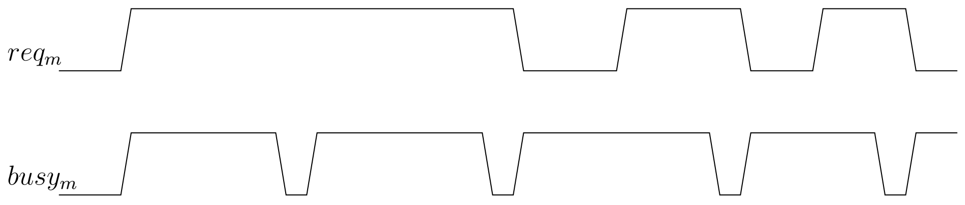

Figure 3.4: Example of Signal $busy_m$ Behaviour

We define the predicate for the *TLB* interface correctness in the following way:

Definition 3.1.14 *We call the TLB correct if the following predicate holds:*

$$\begin{aligned} &good\_tlb\_interface(trc) := \\ &\qquad tlb\_data\_consist(trc) \wedge \\ &\qquad tlb\_purge\_comp(trc) \end{aligned}$$

For the interface between the *MMU* and the memory we make assumptions guaranteeing a correct result of any memory request.

We define a predicate for a memory request similar to the predicate *is_req_proc*.

Definition 3.1.15 *The predicate is_req_mem is defined in the following way:*

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

Note that this predicate slightly differs from *is_req_proc*. In case $t > 0$ a memory request can also start in a cycle t with $busy_m^{t-1} \wedge \neg req_m^{t-1}$(see Figure 3.4), i.e. for this definition the signal *busy* is allowed to be undefined while no requests are generated.

Definition 3.1.16 *We call the memory inputs* live *if the following predicate holds:*

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

This predicate corresponds to the specification of the memory. We define a more precise predicate convenient to use.

Definition 3.1.17 *The predicate mem_liveness_strong(trc) is defined in the following way:*

$$\begin{aligned}
&\mathit{mem\_liveness\_strong}(\mathit{trc}) := \\
&\quad \forall t \in \mathbb{N} : \mathit{req}_m^t \implies \\
&\qquad \exists t' \in \mathbb{Z}_{\geq t} : \neg \mathit{busy}_m^{t'} \wedge \\
&\qquad\quad \forall t'' \in \left[t : t'\right[\ : \ \mathit{busy}_m^{t''}
\end{aligned}$$

One can easily conclude that *mem_liveness_strong(trc)* follows from the predicate *mem_liveness(trc)*.

Lemma 3.1.18 *The implication below trivially holds:*

$$\forall \mathit{trc} : \mathit{mem\_liveness}(\mathit{trc}) \implies \mathit{mem\_liveness\_strong}(\mathit{trc})$$

Since the implication is obvious, we skip the proof of this lemma.

We split the consistency of the memory into three assumptions. The first assumption concerns the page table and the other two cover terminating read and write accesses, respectively. Note that we assume here that the *MMU* has not an exclusive access to the memory.

Definition 3.1.19 *For instants of time $\tau, \tau' \in \mathbb{N}$ as border points on the trace the predicate mem_ack_pte(τ, τ', trc) specifies while the user mode is set the page table entry does not change:*

$$\begin{aligned}
&\mathit{mem\_ack\_pte}(\tau, \tau', \mathit{trc}) := \\
&\quad \forall t', t'' \in \mathbb{N} : \tau \leq \tau' \wedge t' < t'' \wedge t'' \in \left[\tau : \tau'\right] \wedge \\
&\qquad \mathit{req}_p^{t'} \wedge (\forall t \in \left[t' : t''\right] \ : \ \mathit{mode}_p^t) \wedge \\
&\qquad \left\langle \mathit{addr}_p^{t''}[28:9] \right\rangle \leq \left\langle \mathit{ptl}_p^{t''} \right\rangle \implies \\
&\qquad (\mathit{ptea}_p^{t''}[0] \implies \mathit{mem}^{t''}[\mathit{ptea}_p^{t''}[29:1]][63:32] = \\
&\qquad\qquad \mathit{mem}^{t'}[\mathit{ptea}_p^{t''}[29:1]][63:32]) \wedge \\
&\qquad (\neg \mathit{ptea}_p^{t''}[0] \implies \mathit{mem}^{t''}[\mathit{ptea}_p^{t''}[29:1]][31:0] = \\
&\qquad\qquad \mathit{mem}^{t'}[\mathit{ptea}_p^{t''}[29:1]][31:0])
\end{aligned}$$

Recall that $\mathit{addr}_p^{t''}[28:9]$ is the page index *px* and the inequality in the predicate means the *lexcp* does not occur. The times τ, τ' are unnecessary for the local correctness of the *MMU* and included into the predicate only for using the proofs of the *MMU* while verifying the *MU*.

Definition 3.1.20 *The predicate mem_read_consist(trc) holds if at the end of any memory read access the correct data is provided on the output* *from the memory:*

$$\begin{aligned}&mem\_read\_consist(trc) :=\\&\qquad \forall t, t' \in \mathbb{N} : is\_req\_mem(t, t', trc) \wedge mr_m^t \implies\\&\qquad\qquad din_m^{t'} = mem^{t'}[addr_m^t]\end{aligned}$$

Definition 3.1.21 *The predicate mem_write_consist(trc) holds if at the end of any memory write access the memory is updated in the following way:*

$$\begin{aligned}&mem\_write\_consist(trc) :=\\&\qquad \forall t, t' \in \mathbb{N} : is\_req\_mem(t, t', trc) \wedge mw_m^t \implies\\&\qquad\qquad \forall b \in \mathbb{Z}_8 : \left|mem^{t'+1}[addr_m^t]\right|_b =\\&\qquad\qquad\qquad \begin{cases} \left|dout_m^t\right|_b & \textit{if } mbw_m^t[b] \\ \left|mem^{t'}[addr_m^t]\right|_b & \textit{otherwise}\end{cases}\end{aligned}$$

Analogously to the other interfaces we define the predicate for the overall memory interface correctness:

Definition 3.1.22 *We call the memory inputs and configuration correct if the memory is consistent and live:*

$$\begin{aligned}&good\_mem\_interface(trc) :=\\&\qquad mem\_liveness(trc) \wedge\\&\qquad mem\_read\_consist(trc) \wedge\\&\qquad mem\_write\_consist(trc)\end{aligned}$$

Note that $mem\_ack\_pte(trc, \tau, \tau')$ is not present among the parts of $good\_mem\_interface(trc)$ because it appears in predicates defined in the next section.

3.1.4 Guarantees of the MMU

In this section we introduce properties modelling the correct behaviour of the *MMU*. They cover operations performed by the unit as well as restrictions for the *MMU* response.

As it obviously follows, the *MMU* must be able to handle any request it is supplied with and perform the correct response without hanging during the work. Therefore, one of the crucial properties of the unit is *liveness.*

Definition 3.1.23 *The predicate proc_liveness(trc) holds if all of the processor requests terminate in finite time, i.e., after setting read, write or fetch signals on the inputs of the MMU, the MMU eventually releases busy:*

$$\begin{aligned}&proc\_liveness(trc) :=\\&\qquad \forall t \in \mathbb{N} : req_p^t \implies \exists t' \in \mathbb{Z}_{\geq t} : \neg busy_p^{t'}\end{aligned}$$

The *MMU* also has to guarantee that the read and write signals for the memory interface are not set simultaneously.

Definition 3.1.24 *The predicate mem_mr_mw_mutexc(trc) holds when the signals read and write to the memory are mutually exclusive:*

$$\begin{aligned}&mem\_mr\_mw\_mutexc(trc) :=\\&\qquad \forall t \in \mathbb{N} : \neg(mr_m^t \wedge mw_m^t)\end{aligned}$$

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

Definition 3.1.25 *The predicate mem_inputs_stable(trc) holds if all the input signals of the memory interface are stable in case the memory is busy:*

$$\begin{aligned}&mem\_inputs\_stable(trc) :=\\&\qquad \forall t \in \mathbb{N} : req_m^t \wedge busy_m^t \implies\\&\qquad\qquad inputs_m^t = inputs_m^{t+1}\end{aligned}$$

In the rest of the section we provide the definitions of the *MMU* operations which were described above in the section 3.1.2

In case of an untranslated read the *MMU* does not translate the address from the processor. The memory data at the processor address is returned and the exception is not raised.

Definition 3.1.26 *An untranslated read is specified by:*

$$\begin{aligned}&untr\_read(trc) :=\\&\qquad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge\\&\qquad\qquad (mr_p^t \vee fetch_p^t) \wedge \neg mode_p^t \implies\\&\qquad\qquad din_p^{t'} = mem^{t'}[addr_p^t] \wedge \neg excp_p^{t'}\end{aligned}$$

In case of an untranslated write access the data from the processor is written into the memory at the processor address. Depending on the memory byte write signal it can be one, two, four, or eight bytes. As in case of the untranslated write the exception is not raised as well.

Definition 3.1.27 *An untranslated write access is specified by:*

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

Both the translated operations assume probable raising the exception during a processor request. If there is no exception, the result reasonably reflects the memory work. So, for the translated read request, the data of the memory at the physical address is returned.

Definition 3.1.28 *An translated read access is specified by:*

$$\begin{aligned}
&tr\_read(trc) := \\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge \\
&\qquad mem\_ack\_pte(t, t', trc) \wedge \\
&\qquad (mr_p^t \vee fetch_p^t) \wedge mode_p^t \implies \\
&\qquad excp_p^{t'} = tr(t).excp \wedge \\
&\qquad (\neg tr(t).excp \implies \\
&\qquad din_p^{t'} = mem^{t'}[tr(t).pa[31:3]])
\end{aligned}$$

The last operation is the translated write. In case of exception the memory is not modified. Otherwise, the data from the processor is written into the memory at the gained physical address.

Definition 3.1.29 *A translated write access is specified by:*

$$\begin{aligned}
&tr\_write(trc) := \\
&\quad \forall t, t' \in \mathbb{N} : is\_req\_proc(t, t', trc) \wedge \\
&\qquad mem\_ack\_pte(t, t', trc) \wedge \\
&\qquad mw_p^t \wedge mode_p^t \implies \\
&\qquad excp_p^{t'} = tr(t).excp \wedge \\
&\qquad (\neg tr(t).excp \implies \\
&\qquad \forall b \in \mathbb{Z}_8 : \Big| mem^{t'+1}[tr(t).pa[31:3]] \Big|_b = \\
&\qquad\quad \begin{cases} \big| dout_p^t \big|_b & \textit{if } mbw_p^t[b] \\ \Big| mem^{t'}[tr(t).pa[31:3]] \Big|_b & \textit{otherwise} \end{cases})
\end{aligned}$$

Finally we can define the overall correctness of the *MMU*.

Definition 3.1.30 *We call a MMU specification trace correct if and only if it fulfills the following predicate:*

$$\begin{aligned}
&mmu\_guarantee(trc) := \\
&\quad proc\_liveness(trc) \wedge \\
&\quad mem\_mr\_mw\_mutexc(trc) \wedge \\
&\quad mem\_inputs\_stable(trc) \wedge \\
&\quad untr\_read(trc) \wedge untr\_write(trc) \wedge \\
&\quad tr\_read(trc) \wedge tr\_write(trc)
\end{aligned}$$

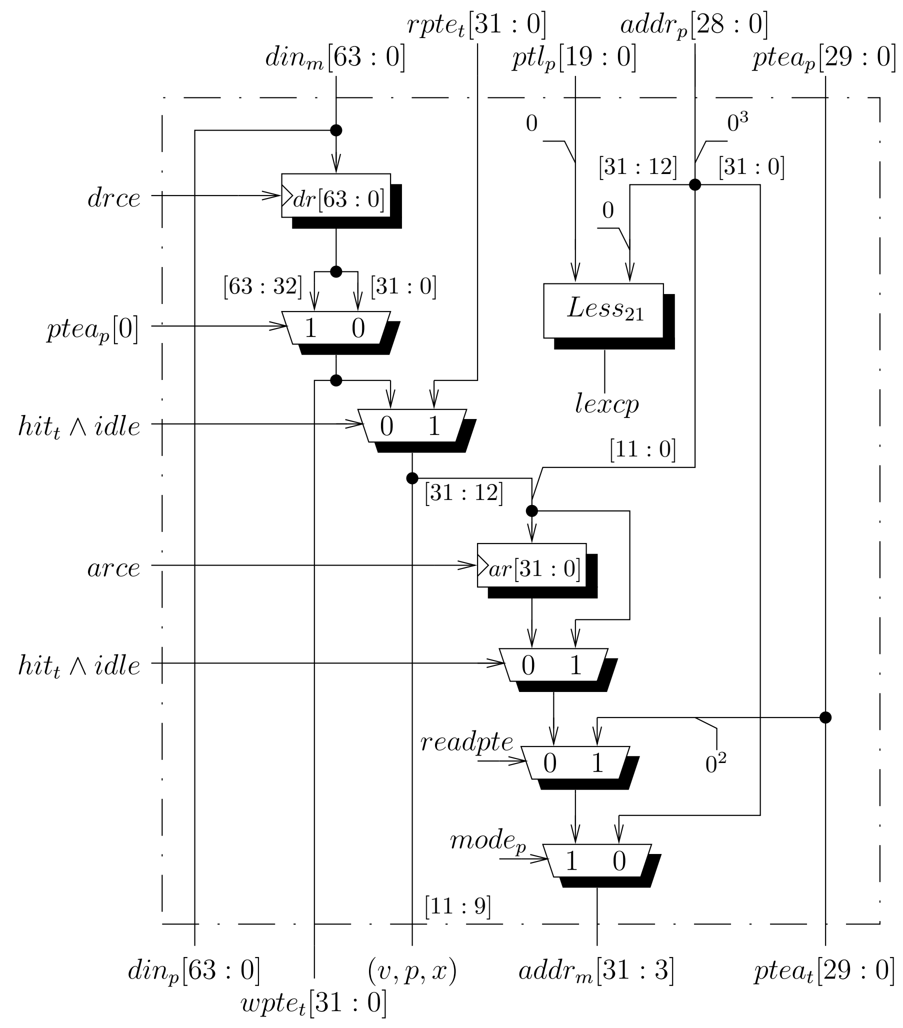

Figure 3.5: Data Paths of *MMU*

3.2 MMU Design

In this section we introduce an implementation of the *MMU*.

Figure 3.5 shows the data paths of the *MMU*. All the components used in the *MMU* construction are covered in Chapter 2. In order to compute the *lexcp* the circuit $Less_{21}$ is used. The data from the memory is saved in the register *dr*. The address register *ar* is used for storing a physical address. Since the memory supports only double word accesses and the page table entry is only one word wide, two page table entries are read at the same time and by using $ptea[0]$ the appropriate one is chosen. Note that the *MMU* must guarantee that the inputs for the memory are stable (Section 3.1.4) during requests as needed by the memory components. Hence, in a digital model

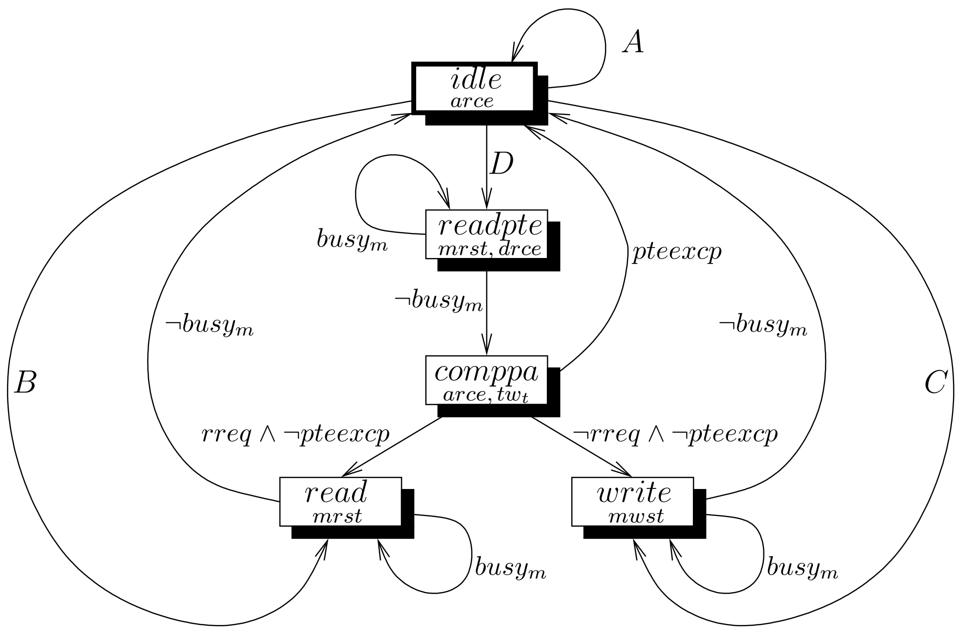

Figure 3.6: Control Automaton of *MMU*

the address $addr_m$ is constant. However, it may not be so as an electrical signal due to glitches when we switch the multiplexers below the register ar. To make $addr_m$ electrically constant we can latch it at the memory side. The control automaton of the *MMU* is presented in Figure 3.6.

The rest of the signals we define using the interface signals and signals from Figure 3.5:

$$\begin{aligned}
mbw_m &= mbw_p && (3.9)\\
dout_m &= dout_p && (3.10)\\
pteexcp &= (mw_p \wedge p) \vee (fetch_p \wedge \neg x) \vee (req_p \wedge \neg v) &&\\
excp &= lexcp \vee pteexcp &&\\
mr_m &= mrst \vee &&\\
&\quad idle \wedge (mr_p \vee fetch_p) \wedge &&\\
&\quad (\neg mode_p \vee mode_p \wedge \neg excp \wedge hit_t) && (3.11)\\
mw_m &= mwst \vee &&\\
&\quad idle \wedge mw_p \wedge (\neg mode_p \vee mode_p \wedge \neg excp \wedge hit_t) && (3.12)\\
tr_t &= req_p \wedge idle && (3.13)\\
ptea_t &= ptea_p && (3.14)\\
excp_p &= mode_p \wedge &&\\
&\quad (lexcp \wedge idle \vee pteexcp \wedge (comppa \vee idle \wedge hit_t)) && (3.15)\\
busy_p &= \neg idle' && (3.16)
\end{aligned}$$

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

In Figure 3.6 for the control automation we use the shorthands:

$$\begin{aligned}
rreq &= mr_p \vee fetch_p \\
A &= \neg req_p \vee \\
&\quad (req_p \wedge mode_p \wedge hit_t \wedge \neg pteexcp \wedge \neg lexcp \wedge \neg busy_m) \vee \\
&\quad (req_p \wedge mode_p \wedge hit_t \wedge pteexcp) \vee \\
&\quad (req_p \wedge mode_p \wedge lexcp) \vee \\
&\quad (req_p \wedge \neg mode_p \wedge \neg busy_m) \\
B &= ((mr_p \vee fetch_p) \wedge mode_p \wedge hit_t \wedge \neg pteexcp \wedge \neg lexcp \wedge busy_m) \vee \\
&\quad ((mr_p \vee fetch_p) \wedge \neg mode_p \wedge busy_m) \\
C &= (mw_p \wedge mode_p \wedge hit_t \wedge \neg pteexcp \wedge \neg lexcp \wedge busy_m) \vee \\
&\quad (mw_p \wedge \neg mode_p \wedge busy_m) \\
D &= req_p \wedge mode_p \wedge \neg hit_t \wedge \neg lexcp
\end{aligned}$$

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

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

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

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

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

Definition 3.2.1 *For the components $c_{mmu}^t.ar$, $c_{mmu}^t.dr$, $c_{mmu}^t.st$ we use abbreviation ar^t, dr^t, and st^t respectively.*

3.3 MMU Correctness

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

For the whole section as before $trc \in Trace$ is a trace induced by the *MMU* implementation.

The correctness of the *MMU* means that if all the assumptions are fulfilled the implementation satisfies the guarantees. Formally, we have to prove the following theorem:

Theorem 3.3.1 *The implementation of the MMU satisfies the specification of the MMU in case the assumptions for all interfaces are fulfilled:*

$$\begin{aligned}&good\_proc\_interface(trc) \wedge good\_tlb\_interface(trc) \wedge \\ &good\_mem\_interface(trc) \implies mmu\_guarantee(trc)\end{aligned}$$

In order to prove this claim we need some intermediate lemmata. First, we prove lemmata following directly from the specification.

Lemma 3.3.2 *During the processor request all of the inputs from the processor are stable:*

$$\begin{aligned}\forall t, t', t'' \in \mathbb{N} : t \leq t' \leq t'' \wedge &is\_req\_proc(t, t'', trc) \wedge \\ &proc\_inputs\_stable(trc) \implies \\ &inputs_p^t = inputs_p^{t'}\end{aligned}$$

Proof: We show the proof by induction on t'. For the base case $t = t'$ the claim is trivially true. For the induction step $t' \to t'+1$ we assume that the statement holds for t' and with this premise one needs to prove the following:

$$\begin{aligned}&t \leq t'+1 \leq t'' \wedge is\_req\_proc(t, t'', trc) \wedge \\ &proc\_inputs\_stable(trc) \implies \\ &inputs_p^t = inputs_p^{t'+1}\end{aligned}$$

From $t \leq t'+1 \leq t''$ the analogous $t \leq t' \leq t''$ does not follow directly. So we consider the cases:

- If $t = t'+1$ then we obviously have the right result.
- If $t < t'+1$ then with the help of the indiction hypothesis we get

  $$inputs_p^t = inputs_p^{t'}$$

  According to the predicate $is\_req\_proc(t, t'', trc)$ for the processor interface there is the signal req_p^t. Therefore, $req_p^{t'}$ is set as well. Moreover, since $t'+1 \leq t''$, we have $t' \neq t''$ and $busy_p^{t'}$. With the help of $proc\_inputs\_stable(trc)$ at the time t' we conclude:

  $$inputs_p^{t'} = inputs_p^{t'+1}$$

This finishes the proof of the lemma. □

One can prove a similar lemma for the memory interface, namely:

Lemma 3.3.3 *All of the inputs in the memory are stable as long as the memory is busy:*

$$\begin{aligned}\forall t, t', t'' \in \mathbb{N} : t \le t' \le t'' \wedge &\, is\_req\_mem(t, t'', trc) \wedge \\ &mem\_inputs\_stable(trc) \implies \\ &inputs_m^t = inputs_m^{t'}\end{aligned}$$

We omit the proof because it is similar to the proof of Lemma 3.3.2.

For simplicity we will use these lemmata further without referring on them. Besides, we assume the control automation of the *MMU* initially is in the state *idle*. The proof of the fact that it is always in one of its states is not worth considering here because it is just bookkeeping.

Now we prove the simple property from Definition 3.1.24, which guarantees that the read and write signals in the memory are mutually exclusive.

Lemma 3.3.4 *Both the signals read and write in the memory can not be active at the same time:*

$$\begin{aligned}&proc\_mr\_mw\_fetch\_mutexc(trc) \\ &\implies mem\_mr\_mw\_mutexc(trc)\end{aligned}$$

Proof: This property follows from the control automation and the computation of the memory read mr_m and memory write mw_m signals according to (3.11) and (3.12).

When the automation is in the states *read* or *readpte* only the signal mr_m is active. In the *write* state the signal mw_m is set but not mr_m. The automation in *comppa* does not generate requests to the memory. As for *idle* state both mr_m and mw_m can be set but not simultaneously because of mutually exclusive signals from the processor as in $proc\_mr\_mw\_fetch\_mutexc(trc)$ predicate.

□

Next we are to prove the property from Definition 3.1.25. At first we will prove only one part of this property, namely, for mr_m.

Lemma 3.3.5 *If in the same cycle both signals mr_m and $busy_m$ are active then mr_m is also active in the next cycle:*

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

Proof: The proof of this lemma also follows from the inspection of the *MMU* control automaton and the computation of the memory read signal mr_m. The signal mr_m can be active in one of the states: *idle*, *read* or *readpte*. For the cycle t we consider the cases:

- The automation in the state *read* or *readpte*. Since the signal $busy_m^t$ is active, the control automation does not change the state in the next cycle. Therefore, mr_m^{t+1} will also be active.
- The automation is in *idle*. Because of active mr_m^t according to the computation of this signal there is $mr_p^t \vee fetch_p^t$ and two variants are possible: either $\neg mode_p^t$ or $mode_p^t \wedge hit_t^t \wedge \neg lexcp^t \wedge \neg ptexcp^t$. Hence, in the next cycle the automation will be in $read^{t+1}$ and mr_m^{t+1} will also be active.

$\square$

The proof for mw_m is analogous, except that the signal can be active only in *idle* and *write* states.

We proceed with the proof of the property from Definition 3.1.25.

Lemma 3.3.6 *The memory inputs are stable:*

$$proc\_inputs\_stable(trc) \implies mem\_inputs\_stable(trc)$$

Proof: From the computation of the memory read mr_m and memory write mw_m signals according to formulae (3.11) and (3.12) one can conclude that in case of a memory request in cycle t the control automation is in one of the states: *idle*, *readpte*, *read*, or *write*.

Let us consider the inputs from the processor side in the cycles t and $t+1$. With respect to the states of the control automation in the cycle t we split the cases:

- The automation is in *idle*. Since req_m^t is active, there is a processor request req_p^t as well. Besides, the two variants are possible: either $\neg mode_p^t$ or $mode_p^t \wedge hit_t^t \wedge \neg lexcp^t \wedge \neg ptexcp^t$. As the memory is busy in t, in the next cycle $t+1$ the control automation will be in *read* or *write* state depending on the processor request. Thus, the signal $busy_p^t$ is set and according to $proc\_inputs\_stable(trc)$ the inputs form the processor are the same for the times t and $t+1$.
- The automation is in *readpte*, *read*, or *write* state. Since it is initially in the state *idle*, we find a cycle t' before t when the processor request starts. The automation leaves *idle* and all the time between t' and $t-1$ the signal $busy_p$ is active. In the cycle t the automation receives $busy_m^t$ from the memory interface side. Consequently, it does not change the state in the next cycle. So, the signal $busy_p^t$ is set as well. With the predicate $proc\_inputs\_stable(trc)$ we obviously get the inputs from the processor do not change between times t' and $t+1$.

Therefore, the data input and the memory byte write input into the memory (both come directly from the processor interface) do not change:

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

As for the memory address, it comes directly from the processor interface or is given by the address register $ar[31:3]$ clocked only in the states *idle* and *comppa*.

If the processor runs in kernel mode, the flag $mode_p$ is not raised at the instants of time t and $t+1$. In this case for both cycles $addr_m = addr_p$ and by the stability shown above it does not change.

If the processor is in user mode some cases appear for the cycle t, namely:

- The automation is in *readpte*. Since $busy_m^t$ is set, the automation does not change the state. The *MMU* is designed so that in case of *readpte* the output address is computed as $addr_m = ptea_p[29:1]$. Since $ptea_p^t = ptea_p^{t+1}$, the memory address remains the same as well.

- The automation is in either *read* or *write*. Therefore, in the next cycle it stays in the same node. For both times the address is $addr_m = ar[31:3]$ and since ar is not clocked in these states, we go to the same conclusion.

- The automation is in *idle*. As it is mentioned above the signal hit_t^t is active. Therefore, the memory address and ar content are computed as:

  $$\begin{aligned} addr_m^t &= (rpte_t^t[31:12] \circ ((addr_p^t \circ 0^3)[11:0]))[31:3] \\ ar^{t+1} &= rpte_t^t[31:12] \circ ((addr_p^t \circ 0^3)[11:0]) \end{aligned}$$

  Since in the next cycle $t+1$ the automation is in *read* or *write*, the memory address is $addr_m^{t+1} = ar^{t+1}[31:3]$. This obviously shows the address $addr_m$ does not change.

So, all the cases concerned with the address for the memory interface prove that the stability holds, namely:

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

For the last two signals mr_m and mw_m we have already proved this property before in Lemma 3.3.5.

$\square$

Lemma 3.3.7 *The processor liveness holds:*

$$\begin{aligned} &mem\_liveness(trc) \wedge proc\_inputs\_stable(trc) \\ &\quad \Longrightarrow proc\_liveness(trc) \end{aligned}$$

Proof: To prove this claim recall the computation of the busy signal for the processor interface according to the equation (3.16). Obviously, the processor interface is live if the *MMU* control automation eventually reaches the state *idle*.

We consider the situation when the processor request signal req_p^t is active. Depending on the current state of the automation in the cycle t the following cases are possible:

- The automation is in either *read* or *write* state. Then we conclude that $mr_m^t \vee mw_m^t$ holds. By $mem\_liveness\_strong(trc)$ we get:

  $$\exists t' \in \mathbb{Z}_{\geq t} : \neg busy_m^{t'} \wedge \forall t'' \in \left[t : t'\right[\ : \ busy_m^{t''}$$

  Therefore, the automation does not change the state for the period from t to t'. And for the next cycle $t'+1$ it reaches *idle*. Hence, $busy_p^{t'}$ is inactive as it must be.

- The automation is in *comppa*. If the exception $pteexcp^t$ occurs, the automation reaches *idle* in $t+1$, and the liveness property holds. Otherwise, the next state is either *read* or *write* and, using the fact shown above, we conclude that $busy_p$ will become inactive for some $t' \in \mathbb{Z}_{\geq t+1}$.

- The automation is in *readpte*. Analogously to the first case, we find $t' \in \mathbb{Z}_{\geq t}$ when $busy_m$ becomes inactive. Therefore, for all $t'' \in [t : t']$ the automation is in *readpte* state and at the time $t'+1$ reaches the state *comppa*. Obviously, $\forall t'' \in [t : t'] : busy_p^{t''}$ and by the predicate $proc\_inputs\_stable(trc)$ the signal $req_p^{t'+1}$ is set as well. Thus, using the previous case we conclude that $busy_p$ will become inactive for some $t''' \in \mathbb{Z}_{\geq t'+2}$.

- The automation is in *idle* state. The signal req_p^t is set and depending on the inputs the automation in the next cycle could be in one of the states: *idle*, *readpte*, *read*, or *write*. In case of *idle* the conclusion is obviously proper. As for the rest of the variants, they are covered by the cases considered above.

This complete inspection of the control automation's work ensures that the liveness property for the processor interface holds indeed.

□

Now we proceed with the correctness proof of all memory operations requested by the processor. But first, we need an auxiliary lemma to be proven:

Lemma 3.3.8 *The following statement is valid:*

$$\forall t \in \mathbb{N}^+ : st^t = idle \implies \neg busy_m^{t-1} \vee \neg req_m^{t-1}$$

Proof: Since in the cycle t the control automation is in the *idle* state, for the time $t-1$ the control must be in one of the states: *comppa*, *read*, *write*, or *idle*. Consider the cases:

- The automation is in the *comppa* state. Here the *MMU* does not generate req_m^{t-1}.
- The automation is in either *read* or *write*. Since the next state is *idle*, the signal $busy_m^{t-1}$ is inactive.
- The control is in *idle*. If the processor is in kernel mode, e.g. $\neg mode_p^{t-1}$, the memory is not busy because the next state is *idle*. In the other case with $mode_p^{t-1}$ being active, if any exception is raised, the request to the memory interface is not produced. Otherwise, the signal $busy_m^{t-1}$ must be inactive to allow the control to reach *idle*.

This all proves the statement. □

Lemma 3.3.9 *Any untranslated read request from the processor to the MMU is performed correctly:*

$$\begin{gathered} proc\_inputs\_stable(trc) \wedge mem\_liveness(trc) \wedge \\ proc\_mr\_mw\_fetch\_mutexc(trc) \wedge mem\_read\_consist(trc) \\ \implies untr\_read(trc) \end{gathered}$$

Proof: Let $t, t' \in \mathbb{N}$ satisfy $is\_req\_proc(t, t', trc)$. Note that in both cycles t and $t'+1$ the control is in *idle* state and in any cycle in between it is not in *idle*. Recall also during the processor request all inputs from the processor side are stable.

First, we can prove that the exception signal $excp_p^{t'}$ is not set. Since $\neg mode_p^t$ holds, it is so in the instant of time t' as well. By the computation of $excp_p^{t'}$ according to the equation (3.15) the exception does not occur.

Next, we are concerned with the proof of the data output, namely:

$$din_p^{t'} = mem^{t'}[addr_p^t]$$

Since the automation is in *idle* at t and the untranslated read request is started by the processor, the signal mr_m^t is active by the equation (3.11). Using Lemma 3.3.8 one can state that the memory request is started at t as well. For the further process two variants are possible:

- The memory is not busy, i.e. $\neg busy_m^t$ holds. Based on the facts above the predicate $is\_req\_mem(t, t, trc)$ is fulfilled and the memory request finishes at the same time t. Thus, the data from the memory is

$$din_m^t = mem^t[addr_m^t]$$

Obviously, the next state of the automation is *idle* and the signal $busy_p^t$ is inactive. From the predicate $is\_req\_proc(t, t', trc)$ one can conclude that $t = t'$ (if $t \neq t'$ then $busy_p^t$ is set). From the construction of the *MMU*:

$$din_p^{t'} = din_m^{t'} = mem^{t'}[addr_m^t] = mem^{t'}[addr_p^t]$$

So, in this case the request from the processor is processed in the right way.

- The memory is busy, i.e. $busy_m^t$ is active. Since the memory is live, from the predicate $mem\_liveness\_strong(trc)$ we find the moment when the memory releases $busy_m$, namely:

  $$\exists t'' \in \mathbb{Z}_{\geq t} : \neg busy_m^{t''} \wedge \forall t''' \in \big[t : t''\big[\ : \ busy_m^{t'''}$$

  Thus, $is\_req\_mem(t, t'', trc)$ holds and the memory request finishes at cycle t''. The date received by the *MMU* is:

  $$din_m^{t''} = mem^{t''}[addr_m^t]$$

  One can easily show for the control:

  $$\forall t''' \in \big]t : t''\big] : st^{t'''} = read$$

  Since all the time from $t+1$ to t'' the automation is in *read* and in the last cycle the flag $busy_m^{t''}$ gets inactive, the control reaches *idle* state and:

  $$\neg busy_p^{t''} \wedge \forall t''' \in \big[t : t''\big[\ : \ busy_p^{t'''}$$

  Therefore, based on the predicate $is\_req\_proc(t, t', trc)$ we get $t' = t''$. Again from the construction of the *MMU* the result is:

  $$din_p^{t'} = din_m^{t'} = mem^{t'}[addr_m^t] = mem^{t'}[addr_p^t]$$

So, the cases above cover the correct behavior of the *MMU* in case of any untranslated read request from the processor side. This finishes the proof.

□

Lemma 3.3.10 *Any untranslated write request from the processor to the MMU is performed correctly:*

$$\begin{aligned} & proc\_inputs\_stable(trc) \wedge mem\_liveness(trc) \wedge \\ & proc\_mr\_mw\_fetch\_mutexc(trc) \wedge mem\_write\_consist(trc) \\ & \qquad\qquad \Longrightarrow untr\_write(trc) \end{aligned}$$

Proof: The proof is similar to the proof of the previous lemma, except some details concerned with the memory update.

Let $t, t' \in \mathbb{N}$ satisfy $is\_req\_proc(t, t', trc)$. Note that in both cycles t and $t' + 1$ the control is in *idle* state and in any cycle in between it is not in *idle*. Recall also during the processor request all inputs from the processor side are stable.

First, we can prove that the exception signal $excp_p^{t'}$ is not set. Since $\neg mode_p^t$ holds, it is so in the time t' as well. By the computation of $excp_p^{t'}$ according to the equation (3.15) the exception does not occur.

Next, we consider how the memory updates at the end of the write processor request. Actually, we want to prove the following:

$$\forall b \in \mathbb{Z}_8 : \left| mem^{t'+1}[addr_p^t] \right|_b = \begin{cases} \left| dout_p^t \right|_b & \text{if } mbw_p^t[b] \\ \left| mem^{t'}[addr_p^t] \right|_b & \text{otherwise} \end{cases}$$

Since the automation is in *idle* at t and the untranslated write request is started by the processor, the signal mw_m^t is active by the equation (3.12). Using Lemma 3.3.8 one can state that the memory request is started at t as well. For the further process two variants are possible:

- The memory is not busy, i.e. $\neg busy_m^t$ holds. Based on the facts above the predicate $is\_req\_mem(t, t, trc)$ is fulfilled and the memory request finishes at the same time t. Thus, the memory update is produced as follows:

  $$\forall b \in \mathbb{Z}_8 : \left| mem^{t+1}[addr_m^t] \right|_b = \begin{cases} \left| dout_m^t \right|_b & \text{if } mbw_m^t[b] \\ \left| mem^t[addr_m^t] \right|_b & \text{otherwise} \end{cases}$$

  Obviously, the next state of the automation is *idle* and the signal $busy_p^t$ is inactive. From the predicate $is\_req\_proc(t, t', trc)$ one can conclude that $t = t'$ (if $t \neq t'$ then $busy_p^t$ is set). According to the equations (3.9) and (3.10) and the datapaths of the *MMU*:

  $$\begin{aligned} mbw_m^t &= mbw_p^t \\ dout_m^t &= dout_p^t \\ addr_m^t &= addr_p^t \end{aligned}$$

  Therefore, the memory update is performed correctly as in the specification of the untranslated write request.

- The memory is busy, i.e. $busy_m^t$ is active. Since the memory is live, from the predicate $mem\_liveness\_strong(trc)$ we find the moment when the memory releases $busy_m$, namely:

$$\exists t'' \in \mathbb{Z}_{\geq t} : \neg busy_m^{t''} \wedge \forall t''' \in \big[t : t''\big[\; : \; busy_m^{t'''}$$

Thus, $is\_req\_mem(t, t'', trc)$ holds and the memory request finishes at cycle t''. In this case the memory is updated as follows:

$$\forall b \in \mathbb{Z}_8 : \left| mem^{t''+1}[addr_m^t] \right|_b = \begin{cases} \left| dout_m^t \right|_b & \text{if } mbw_m^t[b] \\ \left| mem^{t''}[addr_m^t] \right|_b & \text{otherwise} \end{cases}$$

One can easily show for the control:

$$\forall t''' \in \big]t : t''\big] : st^{t'''} = write$$

Since all the time from $t+1$ to t'' the automation is in $write$ and in the last cycle the flag $busy_m^{t''}$ gets inactive, the control reaches $idle$ state and:

$$\neg busy_p^{t''} \wedge \forall t''' \in \big[t : t''\big[\; : \; busy_p^{t'''}$$

Therefore, based on the predicate $is\_req\_proc(t, t', trc)$ we get $t' = t''$.

Again as in the previous case the memory is updated as:

$$\forall b \in \mathbb{Z}_8 : \left| mem^{t'+1}[addr_p^t] \right|_b = \begin{cases} \left| dout_p^t \right|_b & \text{if } mbw_p^t[b] \\ \left| mem^{t'}[addr_p^t] \right|_b & \text{otherwise} \end{cases}$$

This all proves the claim of the lemma. □

Lemma 3.3.11 *Any translated read request from the processor to the MMU is consistent:*

$$\begin{aligned} & proc\_inputs\_stable(trc) \wedge mem\_liveness(trc) \wedge \\ & proc\_mr\_mw\_fetch\_mutexc(trc) \wedge mem\_read\_consist(trc) \wedge \\ & tlb\_data\_consist(trc) \wedge tlb\_purge\_comp(trc) \\ & \qquad \Longrightarrow tr\_read(trc) \end{aligned}$$

Proof: Let $t, t' \in \mathbb{N}$ satisfy $is\_req\_proc(t, t', trc)$. In both cycles t and $t'+1$ the control is in *idle* state and in any cycle in between it is not in *idle*. Recall also during the processor request all inputs from the processor side are stable.

Consider the computation of the exception $lexcp^t$ flag in the datapaths:

$$lexcp^t = Less_{21}(0 \circ ptl_p^t, 0 \circ addr_p^t[28:9])$$

By Definition 2.4.2 we have the exact meaning of *lexcp* flag:

$$lexcp^t = (\langle ptl_p^t \rangle < \langle addr_p^t[28:9] \rangle)$$

This obviously corresponds to the specification in *DecodeITr* function, namely:

$$lexcp^t = tr(t).lexcp$$

The further work of the *MMU* depends on the value of $lexcp^t$ flag.

First, we consider the situation when the exception $lexcp^t$ occurs. In this case the control is in *idle* state at the cycle $t+1$. Based on $is\_req\_proc(t, t', trc)$ one gets $t' = t$. Because of $lexcp^t$, we have $tr(t).lexcp$ as well. This concludes the case since we are not interested in data read from the memory.

Next, if there is no $lexcp^t$ exception, a good deal of case splitting work must be done. At the start t of the translated request the *MMU* sets the read flag tr_t^t for the *TLB* interface according to (3.13) and catches the signal hit_t^t. So, the following cases are possible:

- The flag hit_t^t is inactive. Thus, the *TLB* does not contain an appropriate *pte* addressed with $ptea_t^t = ptea_p^t$. In this case the read operation of the page table must be performed. The control automation changes the state for *readpte* and only there starts reading the memory, i.e. mr_m^{t+1} becomes active. By $mem\_liveness\_strong(trc)$ predicate we find a moment when the memory releases $busy_m$, namely:

$$\exists t'' \in \mathbb{Z}_{\geq t+1} : \neg busy_m^{t''} \wedge \forall t''' \in \left[t+1 : t''\right[\ : \ busy_m^{t'''}$$

  Therefore, the predicate $is\_req\_mem(t+1, t'', trc)$ is fulfilled and the memory read request finishes. The data received is clocked into the data register *dr*. With respect to the *MMU* construction, we get the following:

$$dr^{t''+1} = din_m^{t''} = mem^{t''}[addr_m^{t+1}] = mem^{t''}[ptea_p^{t+1}[29:1]]$$

  So, the data register *dr* contains two adjacent page table entries with addresses distinct in the last bit. Later on, the page table entry to be used is chosen with the MUX select signal *ptea*[0].

Now for the consistency with the specification we need to prove that the received *pte* has not changed since the moment t until now. Instantiating the time in $mem\_ack\_pte(t, t', trc)$ by t and t'', we have to prove the premises inside it. The automation is in *idle* at the start of the request. All the time from $t+1$ to t'' it is in *readpte*, and at $t''+1$ reaches *comppa* state. Hence, we conclude that

$$\forall t''' \in \left[t : t''+1\right[\;\; : \; busy_p^{t'''}$$

So, $req_p^t \wedge (\forall t''' \in [t : t''] \;\; : \;\; mode_p^t)$ holds because of the stability of the inputs from the processor side. The other premises are obvious except $t'' \leq t'$. If to assume that the time $t'' > t'$, we get $busy_p^{t'}$. This contradicts $is\_req\_proc(t, t', trc)$. Therefore, $t'' \leq t'$ holds indeed.

Now using $mem\_ack\_pte(t, t', trc)$ and the the stability of the inputs from the processor, we easily get

$$\begin{aligned}
(ptea_p^t[0] \implies & mem^{t''}[ptea_p^{t+1}[29:1]][63:32] = \\
& mem^t[ptea_p^t[29:1]][63:32]) \wedge \\
(\neg ptea_p^t[0] \implies & mem^{t''}[ptea_p^{t+1}[29:1]][31:0] = \\
& mem^t[ptea_p^t[29:1]][31:0])
\end{aligned}$$

Moreover, using Propositions 2.1.12 and 2.1.13 we can state

$$\begin{gathered}
\mathrm{bin}_{29}(\lfloor \langle ptea_p^t \circ 0^2 \rangle /8 \rfloor) = ptea_p^t[29:1] \\
\langle ptea_p^t \circ 0^2 \rangle \bmod 8 = 0 \iff \neg ptea_p^t[0]
\end{gathered}$$

Therefore, the *pte* read from the memory is the same as in the specification:

$$tr(t).pte = \begin{cases} dr^{t''+1}[31:0] & \text{if } \langle ptea_p^t \circ 0^2 \rangle \bmod 8 = 0 \\ dr^{t''+1}[63:32] & \text{otherwise} \end{cases}$$

All the time from $t+1$ to t'' the control in the state *readpte* and at $t''+1$ reaches *comppa* state, where the physical address must be computed. Since the processor inputs are stable, the following statement is obvious:

$$\langle tr(t).ptea \circ 0^2 \rangle \bmod 8 = 0 \iff \neg ptea_p^{t''+1}[0]$$

In the state *comppa* in $t''+1$ the physical address must be computed. We know that $ptea_p^{t''+1} = ptea_p^t$. The clock signal $arce^{t''+1}$ is active.

So, from the datapaths it is easily seen that the address register is updated in the following manner:

$$\begin{aligned} ar^{t''+2}[31:0] &= tr(t).pte[31:12] \circ (addr_p^{t''+1} \circ 0^3)[11:0] \\ &= tr(t).pte[31:12] \circ (addr_p^{t} \circ 0^3)[11:0] \\ &= tr(t).ppx \circ tr(t).bx \\ &= tr(t).pa \end{aligned}$$

The further trace in the control automation depends on the flag $pteexcp^{t''+1}$. Since the processor signals mr_p, mw_p, $fetch_p$ do not change all the time from t to $t''+1$, and the request is of the read sort, the $pteexcp^{t''+1}$ computation is:

$$\begin{aligned} pteexcp^{t''+1} &= (fetch_p^{t''+1} \wedge \neg x) \vee \neg v \\ &= (fetch_p^{t''+1} \wedge \neg tr(t).pte[9]) \vee \neg tr(t).pte[11] \\ &= tr(t).pteexcp \end{aligned}$$

Now we split cases on the value of $pteexcp^{t''+1}$:

- The exception $pteexcp^{t''+1}$ occurs. In this case the control changes the state on *idle* at cycle $t''+2$. Obviously, the *MMU* releases $busy_p$ signal, i.e.:

  $$\neg busy_p^{t''+1} \wedge \forall t''' \in \left[t : t''+1\right[\;:\; busy_p^{t'''}$$

  Based on the predicate $is\_req\_proc(t, t', trc)$ we get $t' = t''+1$. With the equation (3.15) the signal $excp_p^{t'}$ becomes active. According to (3.6) $tr(t).excp$ holds as well. So,

  $$excp_p^{t'} = tr(t).excp$$

  Since in case of an exception the *MMU* does not read data from the memory, all above concludes the case as we need.

- The exception $pteexcp^{t''+1}$ does not occur. Obviously, $\neg tr(t).excp$ holds as well. Since there is a read or fetch processor request and all inputs are stable, the control automation goes into the state *read* at the next cycle $t''+2$. The *MMU* starts the request to the memory, i.e. $mr_m^{t''+2}$ is set. Since the memory is live, from $mem\_liveness\_strong(trc)$ we find the moment when the memory releases $busy_m$, namely:

  $$\exists t''' \in \mathbb{Z}_{\geq t''+2} : \neg busy_m^{t'''} \wedge \forall \hat{t} \in \left[t''+2 : t'''\right[\;:\; busy_m^{\hat{t}}$$

Thus, $is\_req\_mem(t''+2, t''', trc)$ holds and the memory request finishes at cycle t'''. The date received by the *MMU* is:

$$din_m^{t'''} = mem^{t'''}[addr_m^{t''+2}]$$

One can easily show for the control:

$$\forall \hat{t} \in \left]t''+1 : t'''\right] : st^{\hat{t}} = read$$

Since in the last cycle the flag $busy_m^{t'''}$ gets inactive, the control reaches *idle* state and:

$$\neg busy_p^{t'''} \wedge \forall \hat{t} \in \left[t : t'''\right[\ : \ busy_p^{\hat{t}}$$

Therefore, based on the predicate $is\_req\_proc(t, t', trc)$ we get that $t' = t'''$. From the construction of the *MMU* the output address to the memory is:

$$addr_m^{t''+2} = ar^{t''+2}[31:3] = tr(t).pa[31:3]$$

Hence,

$$din_p^{t'} = din_m^{t'} = mem^{t'}[tr(t).pa[31:3]]$$

Besides, since in the cycle t' the control is in *read* state, according to the equation (3.15) the output signal $excp_p^{t'}$ is inactive, and this is the same as in the specification. So, the facts above completely cover the case.

- The flag hit_t^t is active. Thus, the *MMU* receives the page table entry $rpte_t^t$ addressed with $ptea_t^t = ptea_p^t$ as in (3.14) and prior written into the *TLB* before the hit. With the predicate $tlb\_data\_consist(trc)$ we can find such a moment in the past when the page table entry was saved in the buffer. Using $tlb\_purge\_comp(trc)$ we get:

$$\begin{aligned}\exists \tau \in \mathbb{Z}_t : tw_t^\tau \wedge rpte_t^t = wpte_t^\tau \wedge ptea_p^t = ptea_p^\tau \wedge \\ \forall \tau'' : (\tau'' \in [\tau : t[\implies mode_p^{\tau''}) \wedge \\ (\tau'' \in]\tau : t[\implies \neg(tw_t^{\tau''} \wedge ptea_p^{\tau''} = ptea_p^\tau))\end{aligned}$$

Besides, the data written into the *TLB* was:

$$wpte_t^\tau = \begin{cases} dr^\tau[31:0] & \text{if } ptea_p^\tau[0] \\ dr^\tau[63:32] & \text{otherwise} \end{cases}$$

So, we need to find the content of the data register dr at the cycle τ.

Since the write flag tw_t^τ was active, the automation was in *comppa* state at cycle τ and *readpte* at $\tau-1$. Therefore, $drce^{\tau-1}$ was signalled and the data register content changed as follows:

$$dr^\tau = din_m^{\tau-1}$$

It is easy to prove that if the control is in *readpte* state, then:

$$\exists\tau' \in \mathbb{Z}_{\tau-1} : st^{\tau'} = idle \wedge \forall\tau'' \in \left]\tau' : \tau-1\right] : st^{\tau''} = readpte$$

Therefore, in the next cycle $\tau'+1$ the request to the memory was started and $mr_m^{\tau'+1}$ held. It is obvious that all the time from $\tau'+1$ to $\tau-2$ the memory was busy and then $\neg busy_m^{\tau-1}$ held. Hence, $is\_req\_mem(\tau'+1, \tau-1, trc)$ is valid and according to $mem\_read\_consist(trc)$ the *MMU* read the data:

$$din_m^{\tau-1} = mem^{\tau-1}[addr_m^{\tau'+1}]$$

In the datapaths of the *MMU* the memory address was computed as:

$$addr_m^{\tau'+1} = ptea_p^{\tau'+1}[29:1]$$

Based on the input stability from the processor and $ptea_p^t = ptea_p^\tau$ the data input appears to be:

$$din_m^{\tau-1} = mem^{\tau-1}[ptea_p^t[29:1]]$$

All the time from $\tau-1$ to t the processor runs in user mode and $\neg lexcp^t$ holds. After instantiating the time in $mem\_ack\_pte(t, t', trc)$ by $\tau-1$ and t, the premises inside this predicate are obviously true. Thus, we get:

$$\begin{aligned}(ptea_p^t[0] \implies\ & mem^{\tau-1}[ptea_p^t[29:1]][63:32] = \\ & mem^t[ptea_p^t[29:1]][63:32]) \wedge \\ (\neg ptea_p^t[0] \implies\ & mem^{\tau-1}[ptea_p^{t+1}[29:1]][31:0] = \\ & mem^t[ptea_p^t[29:1]][31:0])\end{aligned}$$

We conclude the page table entry received from the *TLB* is the same *pte* the specification consider:

$$tr(t).pte = rpte_t^t$$

As the control in *idle* state the clock signal $arce^t$ is active. So, from the datapaths it is easily seen that the address register is updated as:

$$\begin{aligned} ar^{t+1}[31:0] &= rpte_t^t[31:12] \circ (addr_p^t \circ 0^3)[11:0] \\ &= tr(t).ppx \circ tr(t).bx \\ &= tr(t).pa \end{aligned}$$

After the page table entry has been received from the *TLB* the computation of *pteexcp* is correct. Since $st^t = idle \wedge hit_t^t$ holds, $rpte_t^t$ is used. With the implementation and the equation (3.5) it is easy to show that:

$$tr(t).pteexcp = pteexcp^t$$

The further process depends on this flag. We split cases:

- The exception occurs, i.e. $pteexcp^t$ holds. The control automation does not change the state and the signal $busy_p^t$ is inactive. Therefore this is a one-cycle processor request with $t' = t$. Obviously, the exception signal $excp_p^{t'}$ is set as well as $tr(t).excp$. This finishes the case.
- There is no exception, i.e. $\neg pteexcp^t$ holds. We proceed in analogy with the proof of the untranslated read. The signal mr_m^t is active by the equation (3.11). Using Lemma 3.3.8 one can state that the memory request is started at t as well. Then, two variants are possible:
  * The memory is not busy, i.e. $\neg busy_m^t$ holds. The next state of the automation is *idle* and the signal $busy_p^t$ is inactive. From the predicate $is\_req\_proc(t, t', trc)$ one can conclude that $t = t'$. Obviously, there is no exception and $\neg tr(t).excp$ is true.
    Since the memory request finishes at the same time t, the data form the memory is

    $$din_m^t = mem^t[addr_m^t]$$

    From the construction of the *MMU*:

    $$\begin{aligned} addr_m^t &= (rpte_t^t[31:12] \circ (addr_p^t \circ 0^3)[11:0])[31:3] \\ &= tr(t).pa[31:3] \end{aligned}$$

    Therefore, the data provided by the *MMU* is:

    $$din_p^{t'} = din_m^{t'} = mem^{t'}[addr_m^t] = mem^{t'}[tr(t).pa[31:3]]$$

    So, in this case the request from the processor is processed in the right way.
  * The memory is busy, i.e. $busy_m^t$ is active. As it is proved for the untranslated read in Lemma 3.3.9, there exists such time t'' when the memory releases $busy_m$ and the *MMU* receives:

    $$din_m^{t''} = mem^{t''}[addr_m^t]$$

Since $t' = t''$, the conclusion on the data output is completely the same as in the previous case.
Analogously to the previous case we have $\neg tr(t).excp$. Besides, since the control is in *read* state at cycle t', according to the equation (3.15) the flag $excp_p^{t'}$ is not set, then

$$excp_p^{t'} = tr(t).excp$$

Hence, this case is proved as well.

As one can see no other cases are possible. So, the consideration above proves the consistency of any translated read request.

□

Lemma 3.3.12 *Any translated write request from the processor to the MMU is consistent:*

$$\begin{aligned}
&proc\_inputs\_stable(trc) \wedge mem\_liveness(trc) \wedge \\
&proc\_mr\_mw\_fetch\_mutexc(trc) \wedge \\
&mem\_read\_consist(trc) \wedge mem\_write\_consist(trc) \wedge \\
&tlb\_data\_consist(trc) \wedge tlb\_purge\_comp(trc) \\
&\qquad\qquad \Longrightarrow tr\_write(trc)
\end{aligned}$$

Proof: The proof of this lemma is very similar to the previous one. The difference is that the computation of *pteexcp* takes into account the protection bit p of a page table entry. Besides, the request from the processor is for writing data into the memory instead of reading. Thus, the control automation reaches the state *write* and never *read*. The proof for updating the memory is analogous to that covered in Lemma 3.3.10.

□

At the final analysis we summarize all of the lemmata from 3.3.4 to 3.3.12 to prove Theorem 3.3.1.

Proof: [Theorem 3.3.1]
By now all the parts of the correctness predicate $mmu\_guarantee(trc)$ are proved in the lemmata from 3.3.4 to 3.3.12. Each of the predicates for the processor, *TLB* and memory interfaces, namely, $good\_proc\_interface(trc)$, $good\_tlb\_interface(trc)$, $good\_mem\_interface(trc)$, provide all the assumptions needed for the lemmata. So, $mmu\_guarantee(trc)$ holds.

□

The theorem proven covers the local correctness of the *MMU*. However, a few other lemmata turn out to be needed for the proofs while integrating the *MMU* into the memory unit and verifying the whole *MU*.

Two lemmata below shows that the *MMU* provides the proper address for the memory.

Lemma 3.3.13 *At the end of any untranslated request from the processor the memory address provided by the MMU is correct:*

$$\begin{aligned}\forall t, t' \in \mathbb{N} : \ & proc\_inputs\_stable(trc) \wedge \\ & is\_req\_proc(t, t', trc) \wedge \\ & \neg mode_p^t \implies addr_m^{t'} = addr_p^t\end{aligned}$$

Lemma 3.3.14 *At the end of any translated request from the processor the memory address provided by the MMU is correct:*

$$\begin{aligned}\forall t, t' \in \mathbb{N} : \ & good\_proc\_interface(trc) \wedge good\_tlb\_interface(trc) \wedge \\ & mem\_liveness(trc) \wedge mem\_read\_consist(trc) \wedge \\ & is\_req\_proc(t, t', trc) \wedge mem\_ack\_pte(t, t', trc) \wedge \\ & mode_p^t \wedge \neg tr(t).excp \implies addr_m^{t'} = tr(t).pa[31:3]\end{aligned}$$

We skip the proofs of both the lemmata because they are done in the way as the address is considered in Lemmata 3.3.11 and 3.3.9

Chapter 4

The VAMP Memory Unit

The work presented in this chapter is devoted to a further development of the VAMP memory unit described by Dalinger [Dal06]. The new extended design of the *MU* includes the new *MMU* (Chapter 3) and supports an interface for external devices [Tve08].

Since the *MU* was not considered as an independent module of the processor in [Dal06], we follow the same line, but focus only on the VAMP specification and construction points crucial for the *MU*.

4.1 Specification of the MU in the VAMP

To specify the memory unit we consider the specification configuration of the whole VAMP. It consists of two programm counters, three register files and the memory. A step of the processor computation is an execution of a single instruction fetched from the memory at an address given by the programm counter.

The programm counters realizes a so-called *delayed PC* mechanism with one delay slot. An instruction is addressed with $DPC \in \mathbb{B}^{32}$, and $PC' \in \mathbb{B}^{32}$ is used to refer the next instruction to be accessed. The detailed definition of this construction is provided by Müller and Paul in [MP00].

The register files of the VAMP are: general purpose registers ($GPR : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$), floating point registers ($FPR : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$), and special purpose registers ($SPR : \mathbb{Z}_{17} \rightarrow \mathbb{B}^{32}$). A description of all *SPR* registers is provided in [Dal06]. We use only three of them, namely *PTO*, *PTL*, and *MODE*, whose content is supplied to the *MU* for the address translation.

Formally, the VAMP specification configuration is:

$$c_S := (c_S.GPR, c_S.SPR, c_S.FPR, c_S.M, c_S.PC', c_S.DPC)$$

Note that $c_S.M \in Memory$. Since the VAMP is extended for accessing external devices, the memory is a partial mapping in contrast to the definition of the memory in [Dal06]. We denote by *MA* a set of addresses where the memory is defined. The processor accesses an external device by performing a word (32-bit) load/store operation at an address associated with this device. The external devices are accessed with 30-bit addresses, the higher 17 bits of which are set to ones. Specifying the processor interface for the external devices and results of accessing them is out of the scope of this thesis and is covered in [Tve08]

By analogy with [Dal06], we consider two specification configurations of the VAMP. The first one c_S can not be affected by interrupts, the second one $\tilde{c}_S$ can change in case an interrupt occurs (indicated by $\mathit{JISR}(\tilde{c}_S)$). We do not concentrate on the VAMP specification step function that can be found in [Dal06] (Chapter 3) as well, but provide only the results of computations concerning the *MU*. Note, that c_S^n (and $\tilde{c}_S^n$) denote configurations before the execution of an instruction n and after executing an instruction $n-1$.

The current mode of the processor is indicated by the lowest bit of the register *MODE*. If it is set to 1, the processor is in *user mode*, otherwise it runs in *kernel mode*. We denote the processor mode by

$$\mathit{mode}(c_S) := c_S.\mathit{SPR}[\mathit{MODE}][0] \tag{4.1}$$

For the page table length and origin we introduce bitvectors of the corresponding registers' content used for the address translation:

$$\mathit{ptl}(c_S) := c_S.\mathit{SPR}[\mathit{PTL}][19:0] \tag{4.2}$$
$$\mathit{pto}(c_S) := c_S.\mathit{SPR}[\mathit{PTO}][19:0] \tag{4.3}$$

If the processor operates in user mode while fetching an instruction, $c_S.\mathit{DPC}$ is treated as a virtual address and the address translation must be performed. We denote by $\mathit{iptea}(c_S)$ the instruction page table address computed as:

$$\mathit{iptea}(c_S) := ((\mathit{pto}(c_S) \circ 0^{12}) +_{32} (0^{10} \circ \mathit{px}(c_S.\mathit{DPC}) \circ 0^2))[31:2] \tag{4.4}$$

where $\mathit{px}(\mathit{va})$ is the page index for a virtual address $\mathit{va} \in \mathbb{B}^{32}$.

Regardless of the processor mode, an instruction address must be aligned properly. Besides, the page table used in the user mode for fetching an instruction is not supposed to reside in the device address space. Otherwise, an exception *instruction misalignment* occurs. We indicate it by the following predicate:

$$\begin{aligned} \mathit{imal}(c_S) \quad := \quad & c_S.\mathit{DPC}[1] \vee c_S.\mathit{DPC}[0] \vee \\ & \mathit{mode}(c_S) \wedge \mathit{iptea}(c_S)[29:1] \notin \mathit{MA} \end{aligned} \tag{4.5}$$

Provided that $c_S.\mathit{DPC}$ is aligned and $\mathit{iptea}(c_S)$ refers the physical memory, in the user mode we use the result of the translation and denote it by the shorthand:

$$\begin{aligned}&\mathit{iDecodeITr}(c_S) := \\ &\qquad \mathit{DecodeITr}(0, 0, 1, \mathit{ptl}(c_S), \\ &\qquad\qquad \mathit{iptea}(c_S), c_S.\mathit{DPC}, c_S.M)\end{aligned}$$

The result of the virtual address translation procedure can be the page fault. If the exception is not raised, we employ the resulting physical address $\mathit{ipa}(c_S)$, that is

$$\mathit{ipa}(c_S) := \mathit{iDecodeITr}(c_S).\mathit{pa} \tag{4.6}$$

Otherwise, the translation is unsuccessful, and the address is considered to be invalid and filled with zeros.

In the processor an exception *instruction page fault* $\mathit{ipf}(c_S)$ is raised if the translation can not complete successfully or a fetch address points out the physical memory.

$$\begin{aligned}\mathit{ipf}(c_S) \quad := \quad & \mathit{mode}(c_S) \wedge (\mathit{iDecodeITr}(c_S).\mathit{excp} \vee \\ & \qquad \mathit{ipa}(c_S)[31:3] \notin \mathit{MA}) \vee \\ & \neg\mathit{mode}(c_S) \wedge c_S.\mathit{DPC}[31:3] \notin \mathit{MA}\end{aligned} \tag{4.7}$$

Definition 4.1.1 *For an adddress addr* $\in \mathbb{B}^{30}$ *we introduce an auxiliary memory function* $\mathit{Mem} : \mathbb{B}^{30} \to \mathbb{B}^{32}$ *defined as:*

$$\mathit{Mem}[\mathit{addr}] := \begin{cases} c_S.M[\mathit{addr}[29:1]][63:32] & \textit{if } \mathit{addr}[0] \\ c_S.M[\mathit{addr}[29:1]][31:0] & \textit{otherwise}\end{cases}$$

Now we can define a function $\mathit{IR}(c_S)$ returning an instruction to be executed in the configuration c_S:

$$\mathit{IR}(c_S) := \begin{cases} \mathit{Mem}[c_S.\mathit{DPC}[31:2]] & \text{if } \neg\mathit{mode}(c_S) \wedge \neg\mathit{imal}(c_S) \wedge \neg\mathit{ipf}(c_S) \\ \mathit{Mem}[\mathit{ipa}(c_S)[31:2]] & \text{if } \mathit{mode}(c_S) \wedge \neg\mathit{imal}(c_S) \wedge \neg\mathit{ipf}(c_S) \\ 0^{32} & \text{otherwise}\end{cases} \tag{4.8}$$

Based on the VAMP instruction set ([Dal06], Appendix A) we can use a set of functions decoding characteristics of the instruction $IR(c_S)$. Speaking about the *MU* we are interested in the following values:

- $\mathit{imm}(c_S)$ - an immediate constant of the instruction $\mathit{IR}(c_S)$,
- $\mathit{RS}1(c_S)$ - contains the index of the first operand (source register) in a register file;
- $\mathit{RD}(c_S)$ - contains the index of the destination register in a register file;
- $mr?(c_S)$ - holds for the instructions with a read access to $c_S.M$,

- $mw?(c_S)$ - holds for the instructions with a write access $c_S.M$,
- $byte?(c_S)$ - holds for the memory instructions with a byte access;
- $word?(c_S)$ - holds for the memory instructions with a word access (reading or storing 32-bit words);
- $half?(c_S)$ - holds for the memory instructions with a half word access;
- $double?(c_S)$ - holds for the memory instructions with a double word access;
- $u?(c_S)$ - holds for the memory instructions with an unsigned read access when the result is zero extended by an operation *zext*.

All data accesses for loading/storing from/to the physical memory or devices are performed with the *effective address* computed as:

$$ea(c_S) := c_S.GPR[RS1(c_S)] +_{32} sext(imm(c_S)), \tag{4.9}$$

where *sext* denotes an operation providing a sign extended result.

For the data page table address we define $dptea(c_S)$ as:

$$dptea(c_S) := ((pto(c_S) \circ 0^{12}) +_{32} (0^{10} \circ px(ea(c_S)) \circ 0^2))[31:2] \tag{4.10}$$

As in case of instruction fetching, for data accesses an exception *data misalignment* is possible. We indicate it with $dmal(c_S)$.

$$\begin{aligned} dmal(c_S) \;\; := \;\; & (\neg byte?(c_S) \wedge ea(c_S)[0]) \vee \\ & (word?(c_S) \wedge ea(c_S)[1]) \vee \\ & (double?(c_S) \wedge (ea(c_S)[1] \vee ea(c_S)[0])) \vee \\ & (mode(c_S) \wedge dptea(c_S)[29:1] \notin MA) \end{aligned} \tag{4.11}$$

If the data misalignment is not raised, in user mode one can use the translation operation result $dDecodeITr(c_S)$:

$$\begin{aligned} & dDecodeITr(c_S) := \\ & \qquad DecodeITr(mr?(c_S), mw?(c_S), 0, ptl(c_S), \\ & \qquad\qquad dptea(c_S), ea(c_S), c_S.M) \end{aligned}$$

Analogously, we define the *data physical address* used for translated accesses to the physical memory or devices and the *data page fault* flag provided by the memory unit:

$$dpa(c_S) \;\; := \;\; dDecodeITr(c_S).pa \tag{4.12}$$

$$\begin{aligned} dpf(c_S) \;:=\;& mode(c_S) \wedge (dDecodeITr(c_S).excp \vee \\ & \qquad dpa(c_S)[31:3] \notin MA \wedge \neg word?(c_S)) \vee \\ & \neg mode(c_S) \wedge ea(c_S)[31:3] \notin MA \wedge \neg word?(c_S) \end{aligned} \tag{4.13}$$

Let d be a number of bytes to be read/written from/to the memory (1, 2, or 4 depending on the instruction). By $dest$ we denote the 64-bit data to be written to the memory.

$$dest = \begin{cases} 0^{32} \circ c_S.GPR[RD(c_S)] & \text{if } byte? \vee word? \vee half? \\ c_S.FPR[RD(c_S)[4:1]1] \circ c_S.FPR[RD(c_S)[4:1]0] & \\ & \text{if } double? \\ 0^{32} \circ c_S.FPR[RD(c_S)] & \text{otherwise} \end{cases} \tag{4.14}$$

Then a result $data(c_S)$ of reading and the next step computation of the physical memory are specified in the following way:

$$data(c_S) := \begin{cases} c_S.M[ea(c_S)[31:3]] & \text{if } double?(c_S) \wedge \neg mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ c_S.M[dpa(c_S)[31:3]] & \text{if } double?(c_S) \wedge mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ twice_{32}\left(zext\left(|Mem[ea(c_S)[31:2]]|_{\langle ea(c_S)[1:0]\rangle, d}\right)\right) & \\ & \text{if } u?(c_S) \wedge \neg double?(c_S) \wedge \neg mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ twice_{32}\left(zext\left(|Mem[dpa(c_S)[31:2]]|_{\langle dpa(c_S)[1:0]\rangle, d}\right)\right) & \\ & \text{if } u?(c_S) \wedge \neg double?(c_S) \wedge mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ twice_{32}\left(sext\left(|Mem[ea(c_S)[31:2]]|_{\langle ea(c_S)[1:0]\rangle, d}\right)\right) & \\ & \text{if } \neg u?(c_S) \wedge \neg double?(c_S) \wedge \neg mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ twice_{32}\left(sext\left(|Mem[dpa(c_S)[31:2]]|_{\langle dpa(c_S)[1:0]\rangle, d}\right)\right) & \\ & \text{if } \neg u?(c_S) \wedge \neg double?(c_S) \wedge mode(c_S) \wedge \\ & \quad \neg dmal(c_S) \wedge \neg dpf(c_S) \\ 0^{64} & \text{otherwise} \end{cases} \tag{4.15}$$

$$c_S.M' := \begin{cases} c_S.M \ \mathit{with} \ c_S.M[ea(c_S)[31:3]] = dest & \\ \qquad \text{if } mw?(c_S) \wedge double?(c_S) \wedge \neg mode(c_S) \wedge \\ \qquad \neg dmal(c_S) \wedge \neg dpf(c_S) & \\ c_S.M \ \mathit{with} \ c_S.M[dpa(c_S)[31:3]] = dest & \\ \qquad \text{if } mw?(c_S) \wedge double?(c_S) \wedge mode(c_S) \wedge \\ \qquad \neg dmal(c_S) \wedge \neg dpf(c_S) & \\ c_S.M \ \mathit{with} \ |Mem[ea(c_S)[31:2]]|_{\langle ea(c_S)[1:0]\rangle,d} = |dest|_{0,d} & \\ \qquad \text{if } mw?(c_S) \wedge \neg double?(c_S) \wedge \neg mode(c_S) \wedge \\ \qquad \neg dmal(c_S) \wedge \neg dpf(c_S) & \\ c_S.M \ \mathit{with} \ |Mem[dpa(c_S)[31:2]]|_{\langle dpa(c_S)[1:0]\rangle,d} = |dest|_{0,d} & \\ \qquad \text{if } mw?(c_S) \wedge \neg double?(c_S) \wedge mode(c_S) \wedge \\ \qquad \neg dmal(c_S) \wedge \neg dpf(c_S) & \\ c_S.M & \text{otherwise} \end{cases} \tag{4.16}$$

4.2 Implementation of the VAMP Memory Unit

Describing the implementation of the VAMP memory unit and arguing about its correctness require covering some of the VAMP design features. So, below we dedicate a few words to the whole processor and proceed with the detailed explanation of the *MU* construction.

4.2.1 The VAMP Design Overview

Figure 4.1 shows the top-level data paths of the VAMP with the out-of-order instruction execution based on the Tomasulo algorithm [Tom67].

Using a special reorder buffer (*ROB*) guarantees that instructions leave the pipeline in the same order they are fetched for the execution. This allows to realize the precise interrupts mechanism, which assures that if an instruction is interrupted, it is executed and written back and all the following instructions are flushed.

Every register in the register file is extended with two fields from the producer table: a valid bit and an instruction tag. A value of the valid bit indicates whether in the pipeline there is an instruction updating a corresponding register. If this value is set inactive, the tag points to the last instruction that is about to modify the register.

The instruction fetch does not belong to the Tomasulo algorithm and starts with loading a 32-bit instruction from the physical memory and saving the results in $S1$ registers: the instruction register $S1.IR$, the exception flag for the misalignment $S1.imal$, and the page fault flag $S1.ipf$. Note, that accessing external devices for fetching is not allowed.

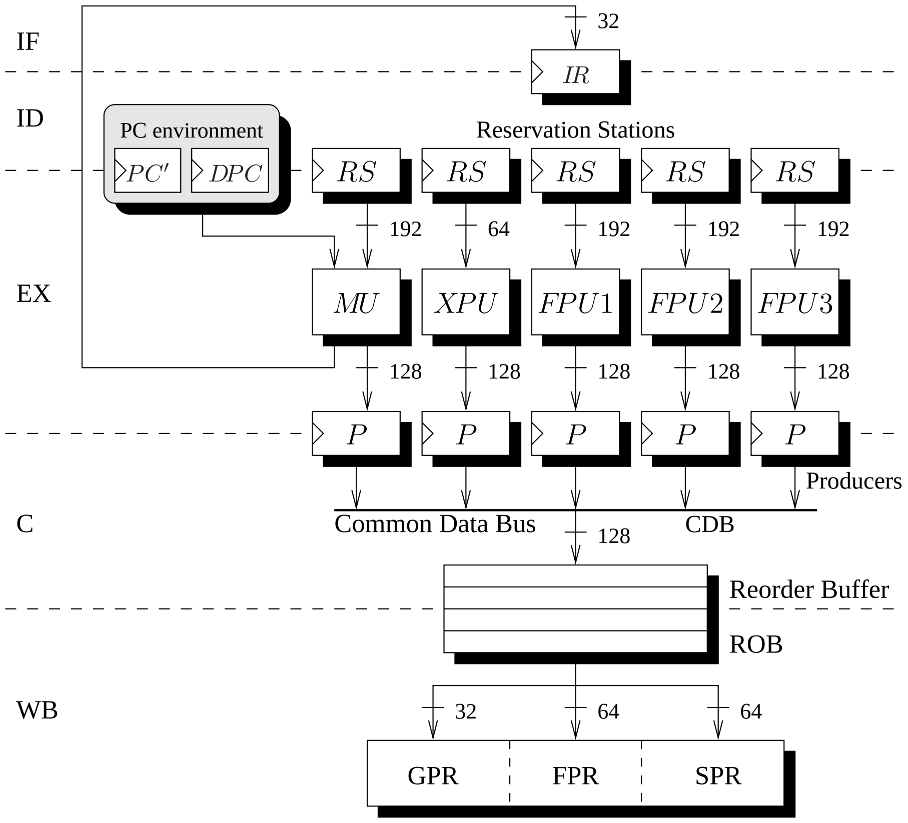

Figure 4.1: Top-level Data Paths of the VAMP

After fetching the instruction is decoded and issued to the reservation station (*RS*) with its source operands. Each functional unit has its own reservation stations. According to [Dal06] the implementation of the Tomasulo algorithm for the VAMP provides eight reservation stations: four *RS*s for the fix point unit and one for each other. Any memory access instruction has up to six source operands: one for the memory address, two 32-bit operands for 64-bit data, and three ones for the page table origin, page table length, and the mode bit. At the end of the issue phase a new non-used tag is reserved for the instruction, its destination registers are marked with this tag and get invalid. Besides, a new entry in the *ROB* is allocated for this instruction.

The instruction is dispatched to the functional unit when all the operands are available and the unit is ready to accept the instruction. Missing operands are catched by snooping the common data bus (*CDB*) connected with units' producers (P). The result of the instruction execution is stored in the corresponding producer and gets available for other instructions waiting for it. Moreover, the instruction with the result appears in the *ROB*. As soon as it becomes the oldest one in the reorder buffer, the result is written back to the register file. So, execution of all instructions affects the register file in order.

There are a few instructions using no functional units. They are described in [Dal06]. More details on the implementation of the Tomasulo algorithm for the VAMP can be found in [Krö01].

Analogously to the specification of the processor, we pay attention only on the VAMP implementation configuration components needed for describing the memory unit. Sticking to the work carried out in [Dal06], we consider the whole processor implementation configuration as a 17-tuple:

$$\begin{aligned} c_I \quad := \quad & (PC', DPC, M, GPR, FPR, SPR, S1, RS, P, ROB, \\ & ROBhead, ROBtail, ROBcount, MU, FPU1, FPU2, FPU3) \end{aligned}$$

ROBhead and *ROBtail* point to the head and the tail of the reorder buffer respectively. *ROBcount* shows the number of entries currently present in the *ROB*. Being referred by the *ROBhead*, an entry of the *ROB* corresponds to the oldest instruction issued and residing in the pipeline.

Through the memory unit the VAMP communicates with off-chip units: the physical memory and the external devices. The component $c_I.M$ includes this memory and the cache system [Bey05, Mül07] of the processor.

To distinguish among the reservation stations and the producers for functional units, we mark them with a subscript containing a name of a particular unit, e.g. RS_{MU} and P_{MU} are used for the memory unit. In the rest of the work instead of writing $c_I.RS_{MU}$, we shorten it as RS_{MU}.

Now we introduce a predicate stating that implementation configurations of the *MU* and some VAMP components needed for arguing about the *MU* are initial.

Definition 4.2.1 *We call an implementation configuration c_I* ***initial*** *for the MU if the memory unit is in an initial state, its reservation station, and the decode stage are empty. Formally, we have:*

$$init(c_I) \quad := \quad initMU(c_I.MU) \wedge \neg c_I.RS_{MU}.valid \wedge \neg c_I.S1.full$$

The predicate $initMU(c_I.MU)$ is described in the next section.

4.2.2 Memory Unit Overall Design

The VAMP memory unit executes all the processor requests to the memory and external devices for instruction fetching and data retrieving. The overall design of the *MU* is depicted on Figure 4.2.

Signals generated by the *MU* are supplied to its subsystems containing the data and instruction *MMU*s (*IMMU* and *DMMU*) and communicating with the off-chip physical memory and external devices via the synchronous bus protocol [MP00, Bey05]. Beside the inputs for the *MMU*s, there are a few signals used for the environment logic. The whole *MMU* environment (*MMU_Env*) is covered in the next section.

Since there is a requirement to guarantee the assumptions for the *MMU*'s inputs (Chapter 3, Section 3.1.3), the new design of the memory unit contains the stabilizing circuits for interrupted memory requests as it was done in [Dal06].

Figure 4.3 gives such a circuit for the instruction fetch. An additional control bit $irollback \in \mathbb{B}$ is active in case an interrupt $JISR(c_I) \in \mathbb{B}$ occurs during a request to *MMU_Env*. It gets inactive in the next cycle after the end of the access to the *IMMU*. Note that the input signal *clear* is actually computed as

$$clear := JISR(c_I) \vee reset \tag{4.17}$$

A register $mPC \in \mathbb{B}^{32}$ is used for storing an instruction fetch address $adr\_i \in \mathbb{B}^{32}$ so that mPC provides the stable address for fetching an instruction in case of an interrupt. The input *iptea* is computed as

$$iptea := Add_{30}(PTO[19:0] \circ 0^{10}, 0^{10} \circ iaddr[28:9], 0)[29:0] \tag{4.18}$$

The request $ifetch$ is only possible if the access address is properly aligned and in user mode *iptea* does not refer to the external devices. The instruction misalignment *imal* is

$$imal := iaddr[1] \vee iaddr[0] \vee AND_{17}(iptea[29:13]) \wedge MODE \tag{4.19}$$

The results of the instruction fetch are saved in the registers of the pipeline stage *IF*, namely $S1$.

Load/store operations are performed in a more complicated way. There is an inner stage m in the *MU* that contains all data and flags required for generating and holding a request to *MMU_Env*. We denote all these registers in the memory unit's configuration as $c_I.MU.m$.

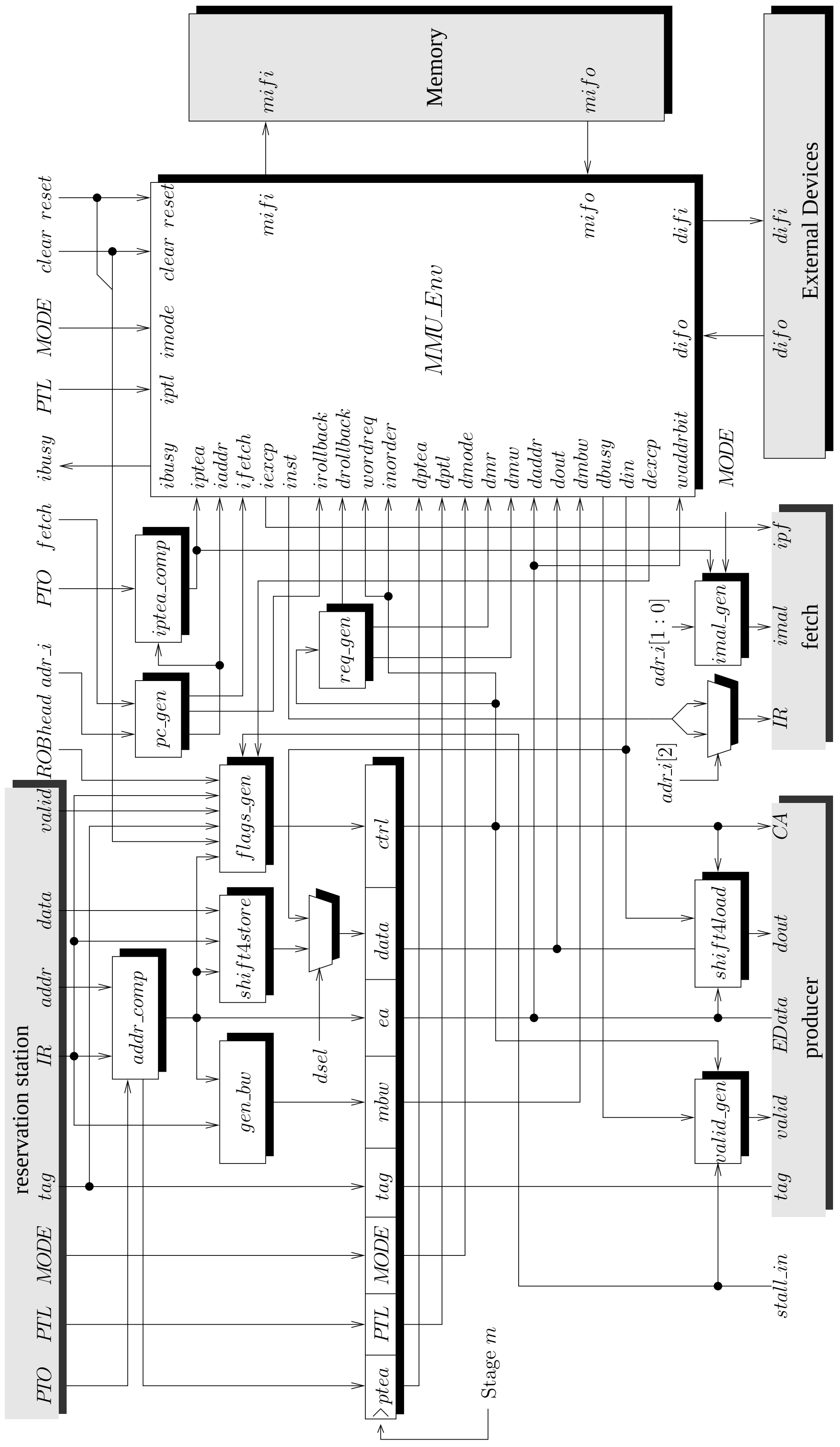

Figure 4.2: The VAMP Memory Unit

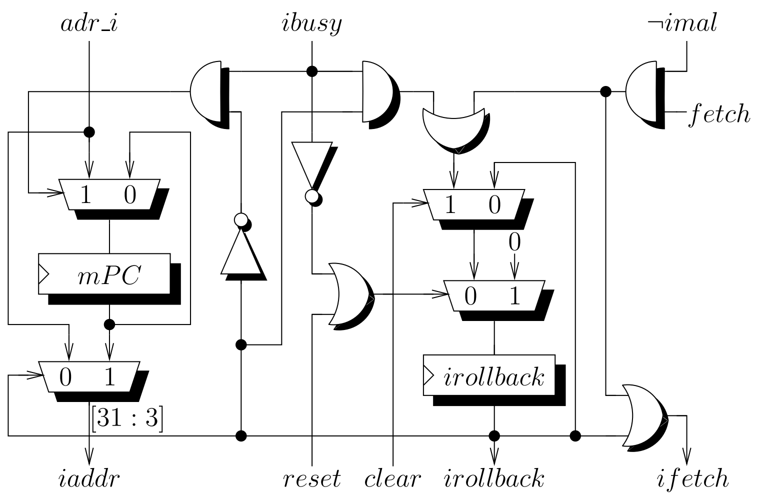

Figure 4.3: Stabilizing Circuit *pc_gen* for the *IMMU*

Definition 4.2.2 *Let r be a register in the stage m of the memory unit. We introduce a short notation m.r for all such registers c_I.MU.m.r.*

A register *m.ctrl* contains a set of the following flags:

$$\begin{aligned} m.ctrl \;:=\; & (valid, stalled, rollback, inorder, \\ & dmal, dpf, Ib, Ih, Iw, Iu, If, Is) \end{aligned}$$

- *rollback* – a flag as the *irollback* flag but used for load/store requests;
- *valid* – a flag indicating that the *MU* processes an instruction and *rollback* is inactive;
- *stalled* – a flag indicating that the producer could not receive the results of an executed instruction in the previous cycle (the Tomasulo scheduler signaled this by active *stall_in*) , the results were buffered and the *MU* was stalled;
- *inorder* – a flag indicating that the current instruction is the oldest one in the *ROB* and a store access to the cache memory system or an access to the devices can be performed;
- *dmal* – a data misalignment flag;
- *dpf* – a data page fault exception received from *MMU_Env*;
- *Ib*, *Ih*, *Iw*, *If* – flags characterizing an operation performed by the current load/store instruction: a byte, half word, word, or a double float access correspondingly;

- *Iu* – a flag indicating that the result of the current load instruction must be zero extended;
- *Is* – a flag of a store instruction.

Definition 4.2.3 *For each flag f in the register $m.ctrl$ we write $m.f$ instead of a record notation $m.ctrl.f$.*

The memory unit receives a new instruction with its operands if only the *MU* does not perform a load/store operation, all the results are provided and not required to be kept in the memory unit further, i.e. a flag *stallout* is inactive.

$$stallout := m.valid \vee m.rollback \tag{4.20}$$

During the instruction dispatch, the tag of the new instruction is copied into the stage m from the reservation station so that

$$m'.tag := \begin{cases} RS_{MU}.tag & \text{if } \neg reset \wedge \neg stallout \\ m.tag & \text{otherwise} \end{cases} \tag{4.21}$$

On the same conditions the registers $PTL \in \mathbb{B}^{20}$, $MODE \in \mathbb{B}$, and the flags *Ib*, *Ih*, *Iw*, *Iu*, *If*, *Is* are updated from the reservation station, e.g. for the flag *Iw*

$$m'.Iw := \begin{cases} Iw(RS_{MU}.IR) & \text{if } \neg reset \wedge \neg stallout \\ m.Iw & \text{otherwise,} \end{cases} \tag{4.22}$$

where *Iw* is a corresponding function. The byte write signals $mbw \in \mathbb{B}^8$ are generated by the circuit *gen_bw*.

The effective address $ea \in \mathbb{B}^{32}$ and the page table entry address $ptea \in \mathbb{B}^{30}$ are computed in *addr_comp* (Figure 4.4) using the reservation station fields: $pto := RS_{MU}.PTO[19:0]$, the sign extended immediate constant $imm := sext(imm(RS_{MU}.IR))$, and an $RS1$-operand *addr*. The computation of *ptea* is taken away from the *MMU* as it was proposed in [Dal06] (Chapter 4) so that the latency of the *MMU* designed in this thesis (Chapter 3) is reduced by one cycle in comparison with the *MMU*'s construction in [Dal06]. However, in contrast to the proposition in that work, using carry save adders [MP00] turned out to be infeasible for the simultaneous computation of both addresses.

The data supplied in case of a store instruction is saved in the register $data \in \mathbb{B}^{64}$ after shifting it in the circuit *shift4store*. This circuit is used for shifting the data that has to be written to a correct byte position in the input double word for the memory cache system.

A circuit *shift4load* is used for the similar shifting as *shift4store* but only performed on the results of read accesses. Since both the signed and unsigned loads are supported in the *MU*, the circuit *shift4load* also produces the sign-extension and zero-extension of the loaded data.

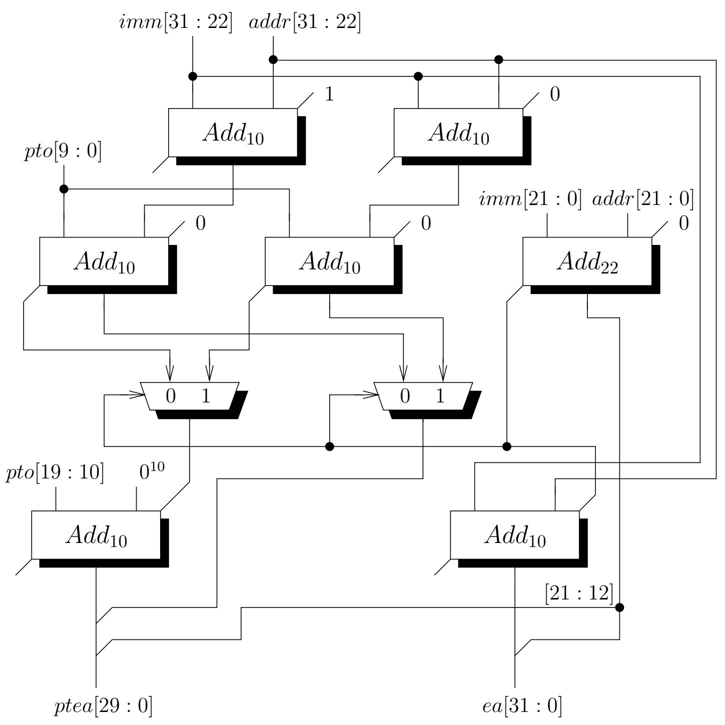

Figure 4.4: Data Paths of *addr_comp*

When the new instruction is dispatched all its operands are ready and the instruction is marked with a set flag $RS_{MU}.valid$. This value is moved to the stage m as well if no interrupts occur. So, the flag *valid* of the stage m is updated during the instruction dispatch and while holding a request.

$$m'.valid := \begin{cases} 0 & \text{if } reset \\ RS_{MU}.valid \wedge \neg clear & \text{if } \neg reset \wedge \neg stallout \\ m.valid \wedge \neg clear & \text{if } \neg reset \wedge \neg stallout \wedge \\ & \quad (clear \vee m.rollback) \\ stall\_in \vee (\neg m.dmal \wedge & \\ \neg m.stalled \wedge dbusy) & \text{otherwise} \end{cases} \tag{4.23}$$

The flags *stalled* and *inorder* are reset but can be changed later too.

$$m'.stalled := \begin{cases} 0 & \text{if } \neg reset \wedge \neg stallout \\ stall\_in & \text{if } \neg reset \wedge stallout \wedge \\ & \quad \neg(clear \vee rollback) \wedge \\ & \quad (\neg dbusy \vee m.dmal \vee \\ & \quad (stall\_in \wedge m.stalled)) \\ m.stalled & \text{otherwise} \end{cases} \tag{4.24}$$

$$m'.inorder := \begin{cases} 0 & \text{if } \neg reset \wedge \neg stallout \\ (ROBhead = m.tag \wedge & \\ \neg m.stalled \wedge \neg m.dmal \wedge & \\ \neg m.inorder) \vee & \\ (m.inorder \wedge dbusy) & \text{if } \neg reset \wedge stallout \wedge \\ & \quad \neg(clear \vee m.rollback) \\ m.inorder & \text{otherwise} \end{cases} \tag{4.25}$$

The data misalignment flag is computed as

$$m'.dmal := \begin{cases} \neg Ib(RS_{MU}.IR) \wedge ea[0] \vee & \\ Iw(RS_{MU}.IR) \wedge ea[1] \vee & \\ If(RS_{MU}.IR) \wedge (ea[1] \vee ea[2]) \vee & \\ AND_{17}(ptea[29:13]) \wedge & \\ RS_{MU}.MODE[0] & \text{if } \neg reset \wedge \neg stallout \\ m.dmal & \text{otherwise} \end{cases} \tag{4.26}$$

A data request to *MMU_Env* starts with generating request signals *dmr* or *dmw* if no data misalignment happened before. Similarly to the instruction fetch, the load/store requests are stabilized in case of $JISR(c_I)$ (Figure 4.5).

$$\begin{aligned} mr &:= m.valid \wedge \neg m.Is \wedge \\ &\quad \neg m.dmal \wedge \neg m.stalled && (4.27) \\ mw &:= m.valid \wedge m.Is \wedge \\ &\quad \neg m.dmal \wedge \neg m.stalled && (4.28) \\ dmr &:= mr \vee (m.rollback \wedge \neg m.Is) && (4.29) \\ dmw &:= mr \vee (m.rollback \wedge m.Is) && (4.30) \end{aligned}$$

Note that now in contrast to [Dal06] a request is started for a store instruction even though the instruction is not the oldest in the pipeline. This is done for eliminating an additional delay because in user mode while the flag *m.inorder* is inactive the *MMU* can perform the address translation. When a write request is generated by the *DMMU*, it waits for the active input *inorder* := *m.inorder* (see also Figure 4.7) and only then it is provided to the memory cache system or devices.

MMU_Env also receives additional signals required for the logic inside: a word request flag *wordreq* := *m.Is* and a word address bit *waddrbit* := *m.ea*[2]. The signal *waddrbit* is provided since the external devices are only word addressable.

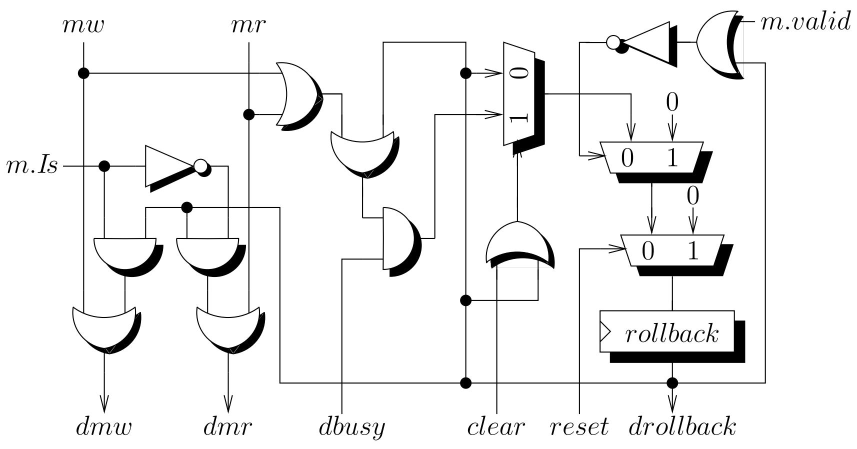

Figure 4.5: Stabilizing Circuit *req_gen* for the *DMMU*

At the end of the data request to *MMU_Env* the results are saved to the stage m namely $m.data$ and $m.dpf$. Simultaneously, the results are provided to the producer in case the signal *stall_in* is not raised. Otherwise, they remain buffered in the stage m until the Tomasulo scheduler can accept them.

Definition 4.2.4 *For any output signal x provided by the memory unit to the producer P_{MU} we introduce a short notation $x_{P_{MU}}$.*

A flag $valid_{P_{MU}}$ generated by the memory unit indicates that an executed instruction leaves it.

$$\begin{aligned} valid_{P_{MU}} &:= \neg stall\_in \wedge m.valid \wedge \\ &\quad (m.dmal \vee m.stalled \vee \neg dbusy) \end{aligned} \tag{4.31}$$

Note, in case of the data misalignment the request is not generated and the result of the instruction execution includes the active flag *m.dmal*.

Now, we can define the predicate $initMU(c_I.MU)$ for the *MU* initial state.

Definition 4.2.5 *The MU is considered to be in the* ***initial*** *state if:*

- *no instruction (that was not interrupted) is executed in the MU;*
- *initial states of both the DMMU and IMMU control automata are unique, i.e. the control can not be in a few states simultaneously (a predicate unique_state holds in this case);*
- *TLBs for both the MMUs are empty, i.e. they do not contain valid entries (a predicate tlb_empty holds in this case).*

Formally, we define the predicate $initMU(c_I.MU)$ as:

$$\begin{aligned}
initMU(c_I.MU) \;:=\; & \neg m.valid \wedge \\
& (\neg m.rollback \implies c_I.MU.DMMU.st = idle) \wedge \\
& (\neg irollback \implies c_I.MU.IMMU.st = idle) \wedge \\
& unique\_state(c_I.MU.DMMU) \wedge \\
& unique\_state(c_I.MU.IMMU) \wedge \\
& tlb\_empty(c_I.MU.DTLB) \wedge \\
& tlb\_empty(c_I.MU.ITLB)
\end{aligned}$$

4.2.3 MMU Environment Construction

The *MMU* environment *MMU_Env* combines both the memory management units with an additional logic and provides accessing the physical memory and external devices via the bus protocol.

The circuit *MMU_Env* is split into two parts corresponding to the *IMMU* and *DMMU*. Both parts communicate with the common instruction/data cache system. Along with caching this system generates inputs $mifi$ for the off-chip memory and processes its respond $mifo$. For managing accesses to the devices and the cache system there is used *prefetch control logic IPFCTRL* and *DPFCTRL*.

Since for the instruction fetch accessing the external devices is not permitted, the construction of the *IMMU* environment (Figure 4.6) is straightforward.

In the *DMMU* environment (Figure 4.7) all load/store word requests to the external devices bypass the cache system because their configuration can change regardless of the processor actions. So, the memory unit communicates directly with the devices according to the synchronous bus protocol. It places a request on $difi$ channel and waits for the result on $difo$ (Figures 4.8 and 4.9). In contrast to the bus protocol described in [MP00], its version considered here supports devices providing the result already in the next cycle after the request [Tve08].

A decision whether an external device or the cache system is accessed is done using an output prefetch control signal $cach\_en_{out}$ computed on basis of an input $addr\_high := addr_m[28 : 12]$ that is the 17 higher bits of the address from the *DMMU*.

Since the *MMU* provides and accepts the signals that correspond to the memory cache system interface and do not match the bus protocol, an additional logic is used for generating signals to external devices and computing the flag *busy* to the *DMMU* side. An external device is considered to be busy in the first cycle of the request and while one of the handshake signals $difi.reqp$ and $difi.brdy$ is raised. As for the output request signals, they are

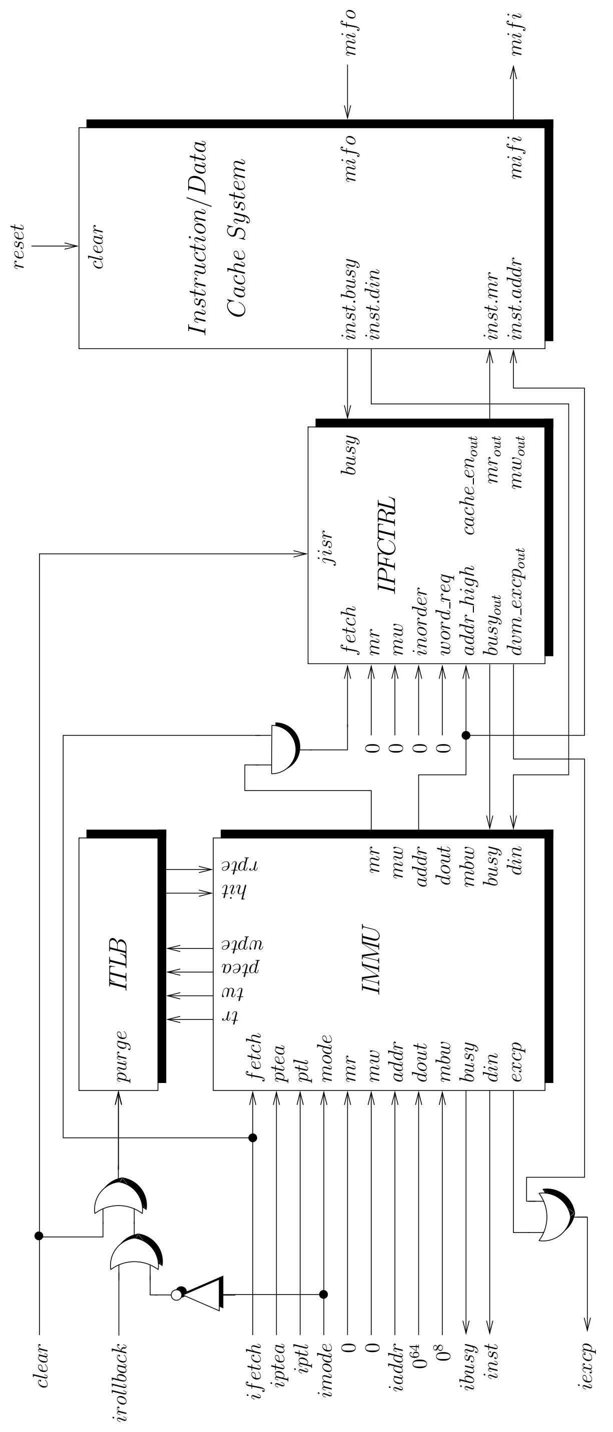

Figure 4.6: *IMMU* Environment

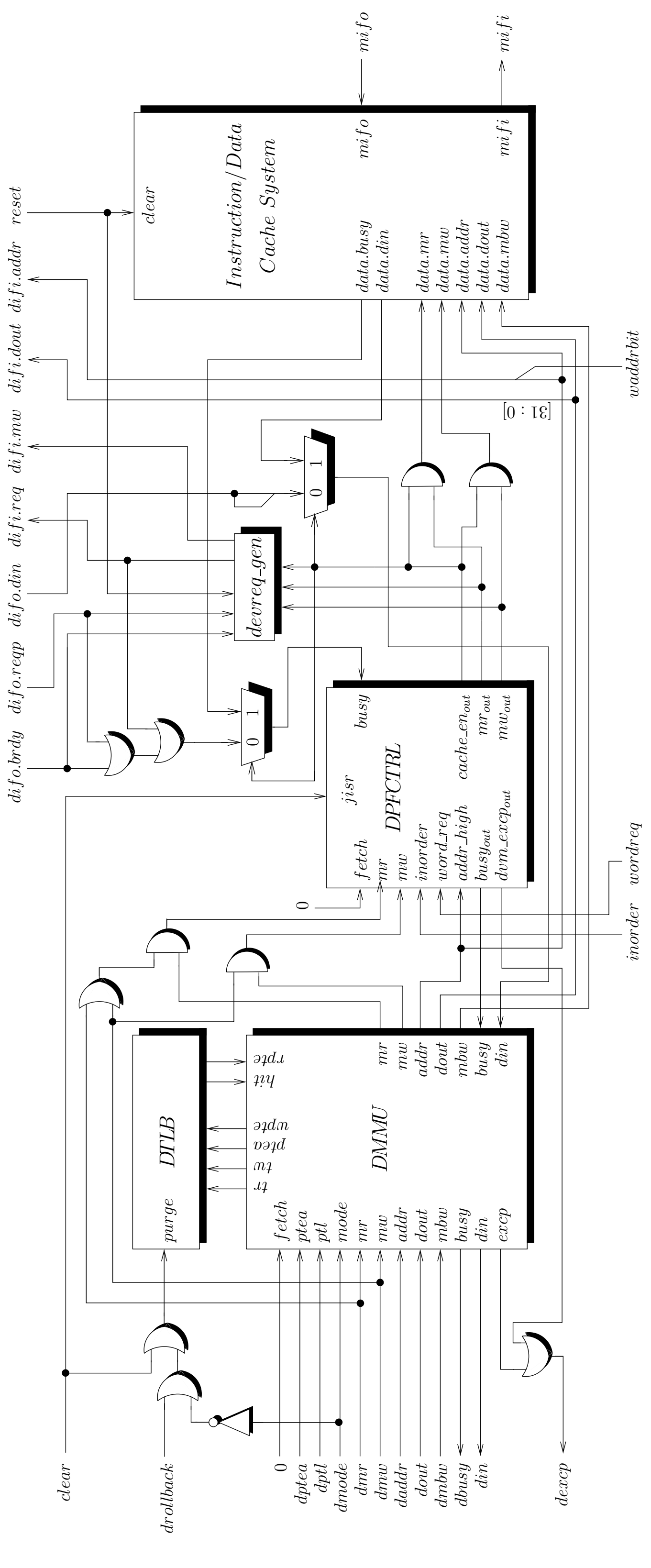

Figure 4.7: *DMMU* Environment

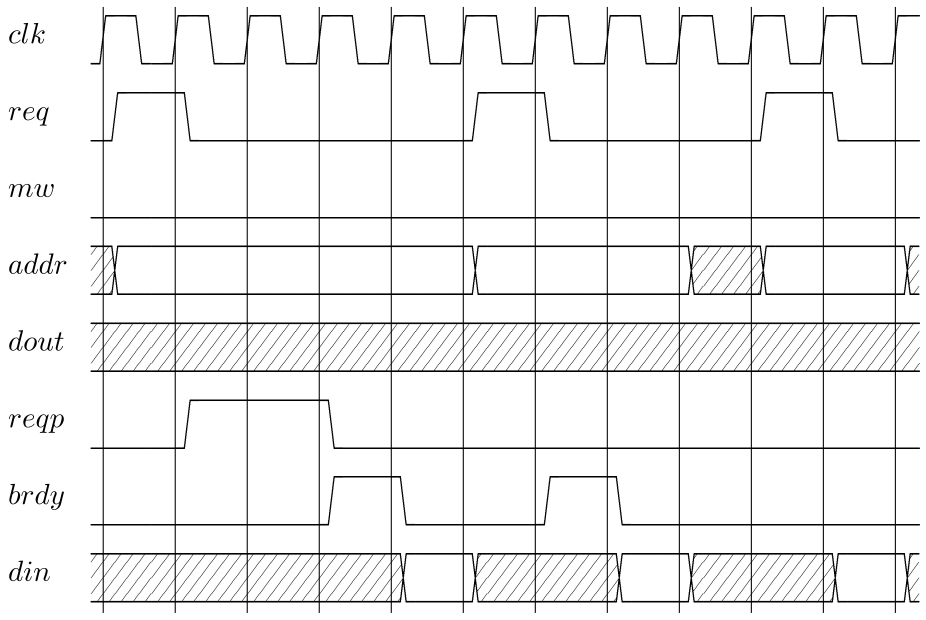

Figure 4.8: Timing of the device interface: read, fast read, and new fast read accesses.

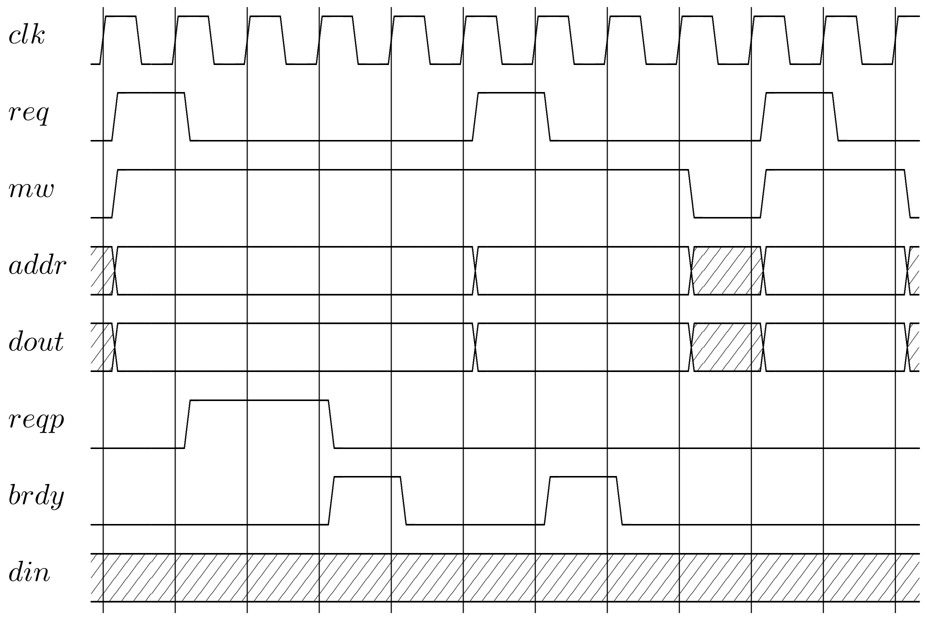

Figure 4.9: Timing of the device interface: write, fast write, and new fast write accesses.

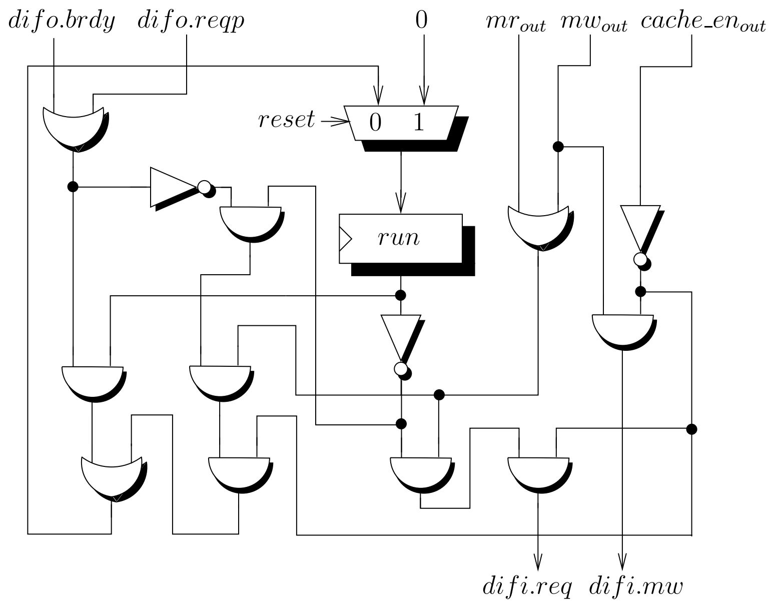

Figure 4.10: Data Paths of *devreq_gen*

computed in a circuit *devreq_gen* (Figure 4.10):

$$difo.req := (mr_{out} \wedge mw_{out}) \wedge \neg run \wedge \neg cache\_en_{out} \tag{4.32}$$

$$difo.mw := mw_{out} \wedge \neg cache\_en_{out} \tag{4.33}$$

The register *run* allows to set *difo.req* only in the first cycle of the request to an external device. It has an active value in case in the pervious cycle the request was in progress and gets 0 after its end.

4.2.4 Prefetch Control Logic

The prefetch control logic circuit *PFCTRL* is designed so that it is used universally for the *IMMU* and *DMMU*. This circuit is included into the *MU* construction because of two reasons. First, the *MU* communicates with external devices. Second, a store instruction can start a request to the *DMMU* even in case the instruction is not the oldest in the pipeline.

In the prefetch control logic a device access is detected and a corresponding flag *dev_mem* is generated inside:

$$dev\_mem := AND_{17}(addr\_high) \tag{4.34}$$

Hence, the output cache enable signal $cache\_en_{out}$ is computed as

$$cache\_en_{out} := \neg dev\_mem \tag{4.35}$$

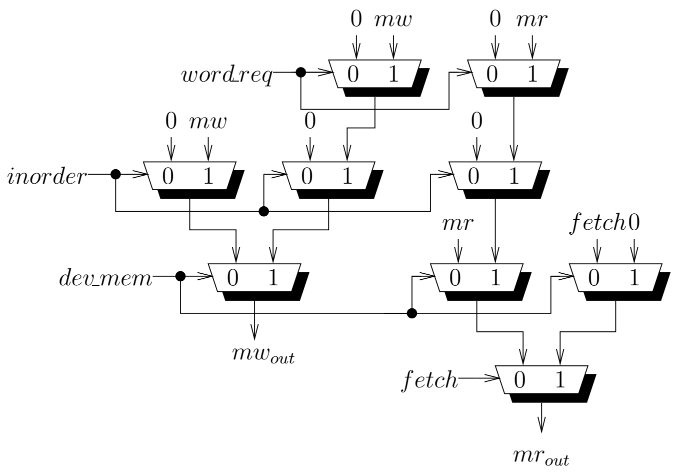

Figure 4.11: Computation of mr_{out} and mw_{out} Signals

Since external devices can be only word accessed and do not provide instructions to be fetched from, an exception dvm_excp_{out} is generated if the *MMU* provides a request intended for an external device and these rules are not abided by.

$$dvm\_excp_{out} := (((mr \vee mw) \wedge \neg word\_req) \vee fetch) \wedge dev\_mem \quad (4.36)$$

In case this exception is raised, neither mr_{out} nor mw_{out} are set to 1 by *PFCTRL*. The same happens if an out-of-order instruction tries to access an external device or to write to the cache memory system. For such cases these outputs remain inactive until the instruction gets the oldest one in the pipeline. The computation of both the signals is depicted on Figure 4.11.

An output signal $busy_{out}$ (Figure 4.12) allows to stall a request $req := mr \vee mw \vee fetch$ from the *MMU* in such a situation with an out-of-order instruction until the proper *inorder* is received. Besides, it gets the value 0 in case of dvm_excp_{out} so that for the *MMU* side it looks like the access to an external device is finished though it is not even started. In all other cases *PFCTRL* just redirects the input *busy* supplied by the cache system or an external device to the *MMU*.

4.3 Correctness Criteria

In this section we provide lemmata to be proven, which cover the correctness of the memory unit. We concentrate only on proofs that show how the local correctness of the *MMU* can be used to verify the entire *MU*.

To organize correctness criteria for the memory unit we follow the same concept used in [Dal06]. The correctness is covered with respect to scheduling

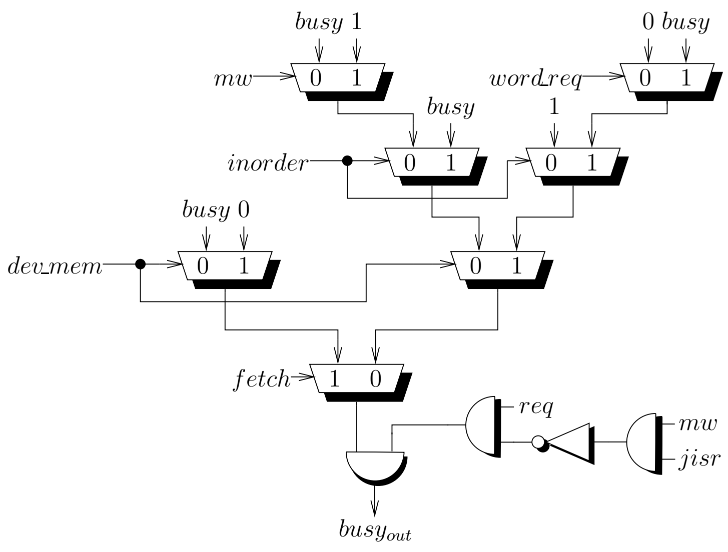

Figure 4.12: Computation of $busy_{out}$ Signal

functions that map a pipeline stage in any cycle to the index of an instruction being in this stage. The scheduling functions were introduced in [MP00, BJK$^+$03]. So, all the previous proofs of VAMP were based on this approach.

4.3.1 Scheduling Functions

A scheduling function $sI(k, t)$ for a stage k is defined inductively over a cycle t. To denote an empty stage initially the scheduling function possesses the value 0, i.e.

$$sI(k, 0) = 0$$

If an instruction in the cycle t is passed from a stage k' to k, the value is changed as

$$sI(k, t + 1) = sI(k', t)$$

Note that the scheduling functions used in VAMP considered in this thesis are analogous to ones covered in [Dal06]. We describe only those crucial for arguing about the *MU* correctness. More details on scheduling functions for visible (included in the programmer's model) and invisible registers can be found in [Bey05, p. 126–132].

The issue scheduling function of the VAMP is incremented on a signal $issue$.

$$\begin{aligned} sI(issue, 0) &:= 1 \\ sI(issue, t+1) &:= \begin{cases} sI(issue, t) + 1 & \text{if } issue^t \\ sI(issue, t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.37}$$

This signal is computed as

$$issue := c_I.S1.full \wedge \neg stall.1 \tag{4.38}$$

and indicates that the instruction register is being issued into some reservation station or directly into the reorder buffer. A flag $c_I.S1.full$ shows that the instruction register contains a fetched instruction. A signal $stall.1$ is the same as introduced in [Bey05] and means that an instruction can not be accepted by the decode stage and issued therefore.

The *MU* reservation station is updated on an active signal $issueRS_{MU}$ computed as follows:

$$\begin{aligned} issueRS_{MU} \;&:=\; c_I.S1.full \wedge \\ &\quad\; \neg(full(ROBcount) \wedge \neg writeback) \wedge \\ &\quad\; \neg RS_{MU}.valid \wedge Imem(c_I.S1.IR), \end{aligned} \tag{4.39}$$

where $full(ROBcount)$ shows that the reorder buffer is full, *writeback* is an update enable signal for the writeback stage, and *Imem* identifies an instruction to be executed in the memory unit. So, the scheduling function for RS_{MU} is defined in the following way:

$$\begin{aligned} sI(RS_{MU}, 0) \;&:=\; 0 \\ sI(RS_{MU}, t+1) \;&:=\; \begin{cases} sI(issue, t) & \text{if } issueRS^t_{MU} \\ sI(RS_{MU}, t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.40}$$

For the inner stage m of the memory unit we use a scheduling function returning an index of an instruction accepted by the the *MU* and executed there in a given instant of time. Dispatching the instruction to the *MU* is allowed on $dispatchRS_{MU}$ flag:

$$dispatchRS_{MU} := RS_{MU}.valid \wedge \neg stallout \tag{4.41}$$

So, the definition of $sI(m, t)$ is straightforward:

$$\begin{aligned} sI(m, 0) \;&:=\; 0 \\ sI(m, t+1) \;&:=\; \begin{cases} sI(RS_{MU}, t) & \text{if } dispatchRS^t_{MU} \\ sI(m, t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.42}$$

For the visible memory $c_I.M$ a scheduling function is defined on basis of $sI(m, t)$ and possess a new value if the execution of an instruction in the memory unit is finished.

$$\begin{aligned} sI(M, 0) \;&:=\; 1 \\ sI(M, t+1) \;&:=\; \begin{cases} sI(m, t) + 1 & \text{if } (mw^t \vee mr^t) \wedge \neg dbusy^t \\ sI(M, t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.43}$$

The last scheduling function required for covering the *MU* correctness concerns the writeback stage:

$$\begin{aligned} sI(wb,0) &:= 1 \\ sI(wb,t+1) &:= \begin{cases} sI(wb,t)+1 & \text{if } writeback^t \\ sI(wb,t) & \text{otherwise} \end{cases} \end{aligned} \tag{4.44}$$

4.3.2 The MU Correctness for Load/Store

The correctness of the memory unit on load /store consists of two points:

- First, at the end of a request the memory $c_I.M$ is updated in the proper way, i.e. the correct date is stored in the memory according to the correct address in case of a store instruction.
- Second, the memory unit produces the same outputs as the corresponding part of the specification.

Both the statements must hold between interrupts. We consider the memory update in more detail. Other claims are just defined to be proven later on.

So far we talked about the memory without covering how it is updated. Using the cache memory system signals depicted on Figure 4.7 we can define the VAMP configuration component $c_I.M$.

Definition 4.3.1 *Let init_mem* $\in$ *Memory be the initial memory content. We introduce a predicate* $bw(ad,b)$ *specifying a write to a byte* $b \in \mathbb{Z}_8$ *at an address* $ad \in \mathbb{B}^{29}$.

$$bw(ad,b) := (ad = data.addr) \wedge data.mw \wedge data.mbw[b] \wedge \neg data.busy$$

The memory content $c_I.M$ *at a cycle* $t \in \mathbb{N}$ *is recursively defined as*

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

Similarly to [Dal06], for the verification effort we use the correctness criteria of the Tomasulo algorithm that guarantees the correctness of the information in the reservation stations.

Definition 4.3.2 *We introduce a predicate* $invRS_{MU}(T)$ *for a* RS_{MU} *invariant stating that the reservation station content in any cycle* $T \in \mathbb{N}$ *corre-*

sponds to an instruction $sI(RS_{MU}, T)$*:*

$$\begin{aligned} invRS_{MU}(T) \;\; := \;\; & RS^T_{MU}.valid \implies \\ & RS^T_{MU}.IR = IR(c_S^{sI(RS_{MU},T)}) \wedge \\ & RS^T_{MU}.PTO = pto(c_S^{sI(RS_{MU},T)}) \wedge \\ & RS^T_{MU}.PTL = ptl(c_S^{sI(RS_{MU},T)}) \wedge \\ & RS^T_{MU}.MODE = mode(c_S^{sI(RS_{MU},T)}) \wedge \\ & RS^T_{MU}.addr = c_S^{sI(RS_{MU},T)}.GPR[RS1(c_S^{sI(RS_{MU},T)})] \end{aligned}$$

Using this invariant allows to conclude the correctness of the register content in the stage m quite easily.

One more note crucial for the *MU* correctness is connected with using the *TLB*. Since the processor allows the data cached in the *TLB* to become inconsistent with the memory content, we introduce a software condition that prohibits a programm running in user mode from modifying the page table.

Definition 4.3.3 *Let* $PT(c_S)$ *be a set of addresses belonging to the page table:*

$$\begin{aligned} startPT(c_S) &:= pto(c_S) \circ 0^{12} \\ endPT(c_S) &:= (pto(c_S) \circ 0^{12}) +_{32} (0^{10} \circ ptl(c_S) \circ 0^2) \\ PT(c_S) &:= \{x \mid \langle x \rangle \in [\langle startPT(c_S) \rangle : \langle endPT(c_S) \rangle]\} \end{aligned}$$

We define a software condition stablePT that must always hold:

$$stablePT \;\; := \;\; \forall i \in \mathbb{N}^+ : mode(c_S^i) \wedge dpa(c_S^i) \in PT(c_S^i) \implies \neg mw?(c_S^i)$$

To use the *MMU* local correctness in the *MU* proofs, we have to define a function composing a *DMMU* trace globally in the memory unit.

Definition 4.3.4 *We introduce a function* $dtrc \in \mathbb{N} \rightarrow Trace$ *computing the local trace* $dtrc^T$ *of the DMMU that starts at a cycle* $T \in \mathbb{N}$ *in the memory unit so that each component returned by* $dtrc^T(t)$ *for a MMU local time* $t \in \mathbb{N}$ *corresponds to a cycle* $T + t$ *in the MU.*

In contrast to [Dal06] the *DMMU* trace used here is not supposed to start only at the beginning of a certain *DMMU* request and concern only this request. We allow it to include a sequence of requests. Later on we show that we can find a cycle for the *MU*, which can be used as an initial cycle for the local *DMMU* trace. These changes are made because the *MMU* exploits the *TLB* in this work.

Note that for the local *DMMU* trace we use the same notation as in Chapter 3. However, we have to be able to denote the *DMMU* signals for each *DMMU* interface at any cycle globally in the memory unit. For example, it is impossible to denote the output exception from the *DMMU* at a cycle $T \in \mathbb{N}$ by the *MMU_Env* output $dexcp^T$ because it includes not only the exception from the *DMMU* but the flag from *DPFCTRL*. To distinguish between the time in the *DMMU* local trace and globally in the memory unit we introduce the following record notation.

Definition 4.3.5 *We use a short notation* $dinputs_x$ *for the DMMU input signals and* $doutputs_x$ *for its output signals in each interface* x *in the memory unit:* p*,* t*, and* m *for the processor, DTLB, and the interface between the DMMU and DPFCTRL. For example,* $doutputs_p^T.excp$ *denotes an output exception from the DMMU at a cycle* $T \in \mathbb{N}$ *in the memory unit.*

We start with auxiliary lemmata used then in final proofs of the *MU* correctness. First, we must prove that the assumptions used for the *MMU* local correctness hold in the memory unit.

Lemma 4.3.6 *The DMMU inputs composing a processor request are correct:*

$$\forall T \in \mathbb{N} : good\_proc\_interface(dtrc^T)$$

We split this lemma in two parts.

Lemma 4.3.7 *The request signals for the DMMU are mutually exclusive:*

$$\forall T \in \mathbb{N} : proc\_mr\_mw\_fetch\_mutex(dtrc^T)$$

Proof: We have to prove that for any time t in the local *DMMU* trace $dtrc^T$ the request signals $fetch_p^t$, mr_p^t, and mw_p^t are mutually exclusive. Actually, for the *DMMU* the signal $fetch_p^t = 0$. The signals mr_p^t and mw_p^t are equal to dmr^{T+t} and dmw^{T+t} in the *MU* correspondingly due to the *DMMU* trace computation. According to (4.29) and (4.30) one easily conclude that these signals are mutually exclusive. □

Lemma 4.3.8 *The DMMU inputs composing a processor request are stable:*

$$\forall T \in \mathbb{N} : proc\_inputs\_stable(dtrc^T)$$

Proof: The inputs for the *DMMU* are supplied from the stage m. The proof follows directly from the memory unit construction since the registers used for the *DMMU* inputs computation are updated only if *dbusy* is inactive. □
Proof: [Lemma 4.3.6]
With both the Lemmata 4.3.7 and 4.3.8 the claim is obvious. □

Lemma 4.3.9 *The DTLB interface is correct:*

$$\forall T \in \mathbb{N} : tlb\_empty(c_I^T.MU.DTLB) \implies good\_tlb\_interface(dtrc^T)$$

As in in the previous case we split the proof of this lemma.

Lemma 4.3.10 *The DTLB is data consistent:*

$$\forall T \in \mathbb{N} : tlb\_empty(c_I^T.MU.DTLB) \implies tlb\_data\_consist(dtrc^T)$$

We omit the proof of this lemma because the *TLB* construction and its correctness are out of the scope of this thesis. This lemma is proven in Isabelle/HOL with respect to the *TLB* design based on the direct mapped cache.

Lemma 4.3.11 *The signal purge for the DTLB is computed correctly in the memory unit:*

$$\forall T \in \mathbb{N} : tlb\_purge\_comp(dtrc^T)$$

Proof: This claim is true with respect to the computation of *purge* in the *DMMU* environment (Figure 4.7)

□

Proof: [Lemma 4.3.9]
With both the Lemmata 4.3.10 and 4.3.11 the claim is obvious.

□

As for the predicate $good\_mem\_interface(dtrc^T)$ for any $T \in \mathbb{N}$, it must hold under the following assumptions:

- the data is read correctly from the memory $c_I.M$, i.e.

$$\begin{aligned} \forall T \in \mathbb{N} : data.mr^T \wedge \neg data.busy^T \implies \\ data.din^T = c_I^T.M[data.addr^T] \end{aligned} \tag{4.45}$$

- the memory $c_I.M$ is live, i.e.

$$\begin{aligned} \forall T \in \mathbb{N} : data.mr^T \wedge data.mw^T \implies \\ \exists T' \in \mathbb{Z}_{\geq T} : \neg data.busy^{T'} \end{aligned} \tag{4.46}$$

- any memory instruction becomes the oldest in the pipeline, i.e.

$$\begin{aligned} \forall T \in \mathbb{N} : dispatchRS_{MU}^T \implies \\ \exists T' \in \mathbb{Z}_{>T} : RS_{MU}.tag^T = ROBhead^{T'} \end{aligned} \tag{4.47}$$

The first assumption is required because we formally defined only $c_I.M$ update. Note that we do not consider requests to the external devices. The other assumptions are used to prove the liveness of the interface between the *DMMU* and *DPFCTRL*. Using the last assumption is caused by the following. As was described in Section 4.2.4, the prefetch control logic stalls a store request to the cache memory system until the memory instruction being executed gets the oldest in the pipeline. Therefore, since the *DMMU* memory interface observation consists of the signals received from and provided for the prefetch control logic, we must guarantee that the flag *m.inorder* is activated eventually. So, (4.47) allows to achieve this in the computation of *m.inorder* according to (4.25). However, to use (4.47) we need the processor liveness that were not proven for the VAMP and must be used as an assumption for the processor now.

With the assumptions above and Lemmata 4.3.6, 4.3.9 we can use Theorem 3.3.1 that covers the local correctness of the *MMU*.

Definition 4.3.12 *We introduce a signal dreq that stands for the data request to the DMMU in the memory unit, namely* $dreq := dmr \vee dmw$.

Now we define a predicate holding only in a cycle when a request to the *DMMU* actually completes.

Definition 4.3.13 *The predicate commits for any* $T, i \in \mathbb{N}$ *is defined in the following way:*

$$commits(T,i) := dreq^T \wedge \neg dbusy^T \wedge m^T.valid \wedge sI(m,T) = i$$

Provided that this predicate holds, we can reconstruct the whole request to the *DMMU* in the past.

Definition 4.3.14 *We define the predicate is_req_proc_dmmu for the processor request to the DMMU in the following way:*

$$\begin{aligned} &is\_req\_proc\_dmmu(T,T') := \\ &\qquad (T > 0 \implies \neg dbusy^{T-1}) \wedge \\ &\qquad (T \le T') \wedge dreq^T \wedge \neg dbusy^{T'} \wedge \\ &\qquad \forall T'' \in [T : T'[\;:\; dbusy^{T''} \wedge \\ &\qquad \forall T'' \in [T : T'] \;:\; m^{T''}.valid \end{aligned}$$

Lemma 4.3.15 *If an instruction in the memory unit commits at a cycle T then there is a cycle* $T' \le T$ *such that from* T' *to T a processor request to the DMMU is performed. Formally, we have:*

$$\begin{aligned} &initMU(c_I^0.MU) \implies \\ &\forall T, i \in \mathbb{N} : commits(T,i) \implies \\ &\qquad \exists T' \in \mathbb{Z}_{\le T} : is\_req\_proc\_dmmu(T',T) \end{aligned}$$

To prove this we first consider a couple of additional lemmata.

Lemma 4.3.16 *If at any cycle T the DMMU is busy and handles a non-interrupted request (as indicated by the signal m.rollback), then there exists a cycle* T' *before T when the DMMU is not busy, all this time till T the* *stage m of the memory unit is valid and the DMMU responds with the active dbusy.*

$$\begin{aligned}
&\mathit{initMU}(c_I^0.\mathit{MU}) \implies \\
&\forall T \in \mathbb{N} : \mathit{dbusy}^T \wedge \neg m^T.\mathit{rollback} \implies \\
&\qquad \exists T' \in \mathbb{Z}_T : \neg \mathit{dbusy}^{T'} \wedge \\
&\qquad\qquad \forall T'' \in \left]T' : T\right] : \mathit{dbusy}^{T''} \wedge m^{T''}.\mathit{valid}
\end{aligned}$$

Proof: We show the claim by induction on T.
Induction base ($\mathbf{T = 0}$): With help of the predicate $\mathit{initMU}(c_I^0.\mathit{MU})$ we have $\neg m^0.\mathit{valid}$. Moreover, for the induction base we get $\neg m^0.\mathit{rollback}$. Therefore, according to Definition 4.3.12, the equations (4.29), and (4.30) we compute $\mathit{dreq}^0 = 0$. However, since dbusy^0 (and in turn busy_p^0 for the trace dtrc^0) is active one can easily conclude that $\mathit{dreq}^0 = 1$. This contradiction completes the case.
Induction step ($\mathbf{T \to T + 1}$): With the induction hypothesis we are to prove the claim:

$$\begin{aligned}
&\exists T' \in \mathbb{Z}_{T+1} : \neg \mathit{dbusy}^{T'} \wedge \\
&\qquad \forall T'' \in \left]T' : T+1\right] : \mathit{dbusy}^{T''} \wedge m^{T''}.\mathit{valid}
\end{aligned}$$

We split cases on $\mathit{dbusy}^T \wedge \neg m^T.\mathit{rollback}$.

- Let $\mathit{dbusy}^T \wedge \neg m^T.\mathit{rollback}$ hold. From the induction hypothesis we have

  $$\begin{aligned}
  &\exists T' \in \mathbb{Z}_T : \neg \mathit{dbusy}^{T'} \wedge \\
  &\qquad \forall T'' \in \left]T' : T\right] : \mathit{dbusy}^{T''} \wedge m^{T''}.\mathit{valid}
  \end{aligned}$$

  Besides, we assume that $\mathit{dbusy}^{T+1} \wedge \neg m^{T+1}.\mathit{rollback}$. Therefore, to prove the statement above it is enough to show that $m^{T+1}.\mathit{valid}$. So far as $\neg \mathit{dbusy}^{T'} \wedge \mathit{dbusy}^{T'+1}$, the request $\mathit{dreq}^{T'+1}$ to the *DMMU* is active. Since the processor inputs are stable from the cycle $T' + 1$ to $T + 1$, we also have dreq^{T+1}. Obviously, one gets $m^{T+1}.\mathit{valid}$ because of $\neg m^{T+1}.\mathit{rollback}$.

- Let $\neg(\mathit{dbusy}^T \wedge \neg m^T.\mathit{rollback})$ hold. We consider two cases depending on the value of dbusy^T.

  – If dbusy^T is active, then $m^T.\mathit{rollback}$ is set, and as a result in the next cycle we get $m^{T+1}.\mathit{rollback}$. However, actually this flag is inactive at $T + 1$. This is a contradiction.

  – If dbusy^T is inactive, the only claim to be proven is $m^{T+1}.\mathit{valid}$. Since $\neg \mathit{dbusy}^T \wedge \mathit{dbusy}^{T+1}$, the request dreq^{T+1} is set. The conclusion is obvious because of $\neg m^{T+1}.\mathit{rollback}$.

□

Lemma 4.3.17 *The flags m.valid and m.rollback in the memory unit are mutually exclusive.*

$$initMU(c_I^0.MU) \implies \neg(m^T.valid \wedge m^T.rollback)$$

Proof: It is easy to prove this assertion by induction on T.
Induction base ($\mathbf{T = 0}$): With the help of the predicate $initMU(c_I^0.MU)$ we have $\neg m^0.valid$. This obviously proves the case.
Induction step ($\mathbf{T \to T + 1}$): To prove the claim

$$\neg(m^{T+1}.valid \wedge m^{T+1}.rollback)$$

it is enough to split the cases on the update conditions for *m.valid* in (4.23). Using the computation of this flag, *m.rollback* update (Figure 4.5), and the induction hypothesis, we easily obtain the appropriate values for $m^{T+1}.valid$ and $m^{T+1}.rollback$. This finishes the proof.

□

With both the lemmata above we can proceed to the proof of Lemma 4.3.15.
Proof: [Lemma 4.3.15]
The claim of the lemma can be proven by induction on T.
Induction base ($\mathbf{T = 0}$): From $commits(0, i)$ we get $m^0.valid$. This contradicts $initMU(c_I^0.MU)$.
Induction step ($\mathbf{T \to T + 1}$): We are to prove that

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

We consider the cases depending on $commits(T, i)$.

- Let $commits(T, i)$ hold. Instantiating the existential quantifier in the assertion with $T + 1$, one sees that all the parts of the predicate $is\_req\_proc\_dmmu(T+1, T+1)$ hold with respect to $commits(T, i) \wedge commits(T + 1, i)$. This covers the case.

- Let $\neg commits(T, i)$ hold. We split the cases on the value of $dbusy^T$:
  - The signal $dbusy^T$ is inactive. With this assumption and $commits(T + 1, i)$ the assertion is obviously true for a one cycle processor request as in the previous case.
  - The signal $dbusy^T$ is active. Since $commits(T + 1, i)$ holds, the flag $m^{T+1}.valid$ is set. Therefore, by Lemma 4.3.17 we get $\neg m^{T+1}.rollback$. Moreover, the input signal $reset^T$ is not active because otherwise we would have $\neg m^{T+1}.valid$. With these all

conditions the computation of $m.rollback$ implies $\neg m^T.rollback$. Now, by Lemma 4.3.16 the following statement holds:

$$\begin{aligned}&\exists t' \in \mathbb{Z}_T : \neg dbusy^{t'} \wedge\\ &\qquad\qquad \forall t'' \in \left]t' : T\right] : dbusy^{t''} \wedge m^{t''}.valid\end{aligned}$$

Hence, instantiating the existential quantifier of the original assertion with $t' + 1$, we obtain all the parts of the predicate $is\_req\_proc\_dmmu(t' + 1, T + 1)$ except $dreq^{t'+1}$. From the statement above we obtain $\neg dbusy^{t'} \wedge dbusy^{t'+1}$. This is true indeed if only $dreq^{t'+1}$ is set. So, the case is completely covered and the proof is finished.

□

To use the local correctness of the *MMU* we must turn the processor request found in the memory unit into the form suitable for the local *MMU* trace. For this purpose we provide a lemma.

Lemma 4.3.18 *The following implication is valid:*

$$\begin{aligned}&\forall t, T, T' \in \mathbb{N} : t \leq T \wedge T \leq T' \wedge\\ &\qquad\qquad is\_req\_proc\_dmmu(T, T') \implies\\ &\qquad\qquad is\_req\_proc(T - t, T' - t, dtrc^t)\end{aligned}$$

We skip the proof of this lemma because it is obvious and requires only expanding the definitions.

Next, we consider a few lemmata concerning the *MU* correctness. Note that all the *DMMU* inputs and the registers of the stage m except $m.inorder$ are stable during a processor request in the memory unit. It is easily proven with exploring the *MU* construction and the proofs are omitted. The same is true for the scheduling function of this stage.

Lemma 4.3.19 *The addresses $m.ptea$ and $m.ea$ are computed correctly in the memory unit, namely for a cycle $T \in \mathbb{N}$*

$$\begin{aligned}&invRS_{MU}(T) \wedge \neg stallout^T \wedge m^{T+1}.valid \implies\\ &m^{T+1}.ea = ea(c_S^{sI(m,T+1)})\\ &m^{T+1}.ptea = dptea(c_S^{sI(m,T+1)})\end{aligned}$$

Proof: The computation of the flag $m^{T+1}.valid$ (4.23) implies $RS_{MU}^T.valid$ because of $\neg stallout^T \wedge m^{T+1}.valid$. Therefore, the condition $dispatchRS_{MU}^T$ for the scheduling function $sI(m, T + 1)$ holds by (4.41) and $sI(m, T + 1) = sI(RS_{MU}, T)$.

The signal $\neg stallout^T$ also indicates that the values of $m^{T+1}.ea$ and $m^{T+1}.ptea$ are supplied from the circuit $addr\_comp$ in the memory unit.

With the help of $invRS_{MU}(T)$ all the inputs for the circuit *addr_comp* correspond to their equivalents in the specification. The correctness of the computations in *addr_comp* is proven with the help of an integrated to Isabelle command [Tve08] calling the model checker SMV. So, the assertion of the lemma is true indeed.

$\square$

Lemma 4.3.20 *The data misalignment flag is computed correctly in the memory unit at the end $T \in \mathbb{N}$ of the processor request. Formally, the following holds:*

$$\begin{aligned}&initMU(c_I^0.MU) \wedge (\forall t \in \mathbb{Z}_T : \neg reset^t \wedge \neg JISR(c_I^t)) \wedge\\&(\forall t \in \mathbb{Z}_T : invRS_{MU}(t)) \wedge is\_req\_proc\_dmmu(T', T) \implies\\&m^T.dmal = dmal(c_S^{sI(m,T)}).\end{aligned}$$

Proof: The stability of the stage m allows to conclude $m^T.dmal = m^{T'}.dmal$. Moreover, during the processor request the memory unit does not accept a new instruction, i.e. $sI(m, T) = sI(m, T')$. So, the assertion is rewritten in the following way:

$$m^{T'}.dmal = dmal(c_S^{sI(m,T')}) .$$

The processor request $is\_req\_proc\_dmmu(T', T)$ implies $\neg stallout^{T'-1}$ and $m^{T'}.valid$. Therefore, the flag $RS_{MU}^{T'-1}.valid$ is set, and in turn $sI(m, T') = sI(RS_{MU}, T' - 1)$ as in the previous lemma. Using the invariant $invRS_{MU}(T' - 1)$, Lemma 4.3.19, *m.dmal* computation by (4.26), and the definition of *dmal* in the specification, we finish the proof of the lemma.

$\square$

Lemma 4.3.21 *If at a cycle $T \in \mathbb{N}$ the memory unit executes an instruction not interrupted before, then there exists a moment in the past that can be considered as an initial cycle of the local DMMU trace, i.e.*

$$\begin{aligned}&initMU(c_I^0.MU) \wedge (\forall t \in \mathbb{Z}_T : \neg reset^t \wedge \neg JISR(c_I^t)) \wedge m^T.valid \implies\\&\exists t \in \mathbb{Z}_T : c_I^t.MU.DMMU.st = idle \wedge tlb\_empty(c_I^t.MU.DTLB) \wedge\\&\qquad (\forall t' \in \mathbb{Z}_t : m^{t'}.rollback) \wedge (\forall t' \in [t : T] : \neg m^{t'}.rollback)\end{aligned}$$

Proof: First, we start with the part of the lemma concerning *m.rollback*:

$$\exists t \in \mathbb{Z}_T : (\forall t' \in \mathbb{Z}_t : m^{t'}.rollback) \wedge (\forall t' \in [t : T] : \neg m^{t'}.rollback)$$

The proof proceeds depending on the following cases:

- $\exists t \in \mathbb{Z}_T : m^t.rollback \wedge (\forall t' \in \ ]t : T] : \neg m^{t'}.rollback)$
  Since there are no interrupts, the flag $m^t.rollback$ can be set only if it is active initially. Therefore, the claim above holds.

- $\forall t \in \mathbb{Z}_T : \neg m^t.rollback \lor (\exists t' \in\ ]t : T] : m^{t'}.rollback)$
  The next case splitting is based on $\forall t \in \mathbb{Z}_T : \neg m^t.rollback$:
  - $\forall t \in \mathbb{Z}_T : \neg m^t.rollback$ holds. This trivially shows the claim is true.
  - Otherwise, one considers $\exists t \in \mathbb{Z}_T : m^t.rollback$. The case implies $\exists t' \in\ ]t : T[\ : m^{t'}.rollback$. Then we can find a cycle with $m.rollback$ and $\neg m.rollback$ after. Moreover, it must be set starting from the initial configuration. This draws the conclusion.

Provided the claim holds, the rest of the lemma to be proven is

$$c_I^t.MU.DMMU.st = idle \land tlb\_empty(c_I^t.MU.DTLB)\ .$$

If $t = 0$ the conclusion is obvious because of $initMU(c_I^0.MU)$. Otherwise, the proof is in exploring the *MU* construction. We already know that $m^{t-1}.rollback$. Hence, the signal $purge^{t-1}$ is raised, and $tlb\_empty(c_I^t.MU.DTLB)$ holds. The flag $m.rollback$ changes its value so that $m^t.rollback$ holds. Using (4.23) we find out $\neg dbusy^{t-1}$. Therefore, the *DMMU*'s control automation is in the state *idle* in the next cycle. This finishes the proof of the lemma.

□

Before considering the memory update we provide a lemma borrowed from [Bey05]. It deals with the memory in the VAMP specification in case a memory instruction commits in the implementation.

Lemma 4.3.22 *If an instruction in the memory unit terminates its access in cycle $T \in \mathbb{N}$, 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*

$$(mw^T \lor mr^T) \land \neg dbusy^T \implies c_S^{sI(M,T)}.M = c_S^{sI(m,T)}.M$$

The proof of this lemma can be found in [Bey05, p. 148]. Note that we translated a large amount of proofs concerning this lemma from the previous PVS code into an Isabelle/HOL theory with slight modifications that do not affect the pencil-and-paper proof.

The proofs of the lemmata and theorems described above in this thesis were carried out and finished in Isabelle/HOL. The correspondence between them and the proofs in Isabelle/HOL is given in Appendix A.

Now we pay attention on lemmata that are not complete in Isabelle/HOL. We just provide a way for further reasoning about the verification of the whole *MU*. The proof below shows how to use the *MMU* local correctness for proving the *MU* correctness.

Lemma 4.3.23 *The exception flag from the DMMU is correct in the MU at a cycle $T \in \mathbb{N}$ when a memory instruction being executed commits. Formally, the following holds:*

$$\begin{aligned}
&\mathit{initMU}(c_I^0.\mathit{MU}) \wedge (\forall t \in \mathbb{Z}_T : \neg \mathit{reset}^t \wedge \neg \mathit{JISR}(c_I^t)) \wedge \\
&(\forall t \in \mathbb{Z}_T : \mathit{invRS}_{\mathit{MU}}(t)) \wedge (\forall t \in \mathbb{Z}_{\leq T} : c_I^t.M = c_S^{sI(M,t)}.M) \wedge \\
&\mathit{commits}(T, sI(m,T)) \implies \\
&(\neg \mathit{dmode}^T \implies \neg \mathit{doutputs}_p^T.\mathit{excp}) \wedge \\
&(\mathit{dmode}^T \implies \mathit{doutputs}_p^T.\mathit{excp} = \mathit{dDecodeITr}(c_S^{sI(m,T)}).\mathit{excp})
\end{aligned}$$

Proof: Using Lemma 4.3.15 we get

$$\exists T' \in \mathbb{Z}_{\leq T} : is\_req\_proc\_dmmu(T', T)$$

With the stability of the stage m during the processor request we have

$$\mathit{dmode}^T = \mathit{dmode}^{T'} = m^{T'}.\mathit{MODE}$$

For the further proof we use the lemmas of the *MMU* local correctness. So, we need to create a local *DMMU* trace in the memory unit. We fix a starting cycle for the local trace according to Lemma 4.3.21, namely

$$\begin{aligned}
\exists t \in \mathbb{Z}_{T'} : c_I^t.\mathit{MU.DMMU.st} = \mathit{idle} \wedge tlb\_empty(c_I^t.\mathit{MU.DTLB}) \wedge \\
(\forall t' \in \mathbb{Z}_t : m^{t'}.\mathit{rollback}) \wedge (\forall t' \in [t : T'] : \neg m^{t'}.\mathit{rollback})
\end{aligned}$$

Therefore, the local *DMMU* trace to be used is dtrc^t.

Now, we can split cases depending in the value of dmode^T.

- The processor runs in kernel mode, i.e. $\neg \mathit{dmode}^T$.

  According to Lemmata 3.3.9 and 3.3.10 we have

  $$\begin{aligned}
  &good\_proc\_interface(\mathit{dtrc}^t) \wedge \\
  &good\_mem\_interface(\mathit{dtrc}^t) \implies \\
  &\forall t', t'' \in \mathbb{N} : is\_req\_proc(t', t'', \mathit{dtrc}^t) \wedge \\
  &\qquad\qquad \mathit{req}_p^{t'} \wedge \neg \mathit{mode}_p^{t'} \implies \neg \mathit{excp}_p^{t''}
  \end{aligned}$$

  The premises concerning the *MMU* interfaces hold by Lemma 4.3.6 and considering $good\_mem\_interface(\mathit{dtrc}^t)$ as it was done before. By Lemma 4.3.18 there is a request $is\_req\_proc(T'-t, T-t, \mathit{dtrc}^t)$ for the local *DMMU* trace. From the *MU* construction one computes

  $$\begin{aligned}
  \mathit{req}_p^{T'-t} &= \mathit{dreq}^{T'} \\
  \mathit{mode}_p^{T'-t} &= m^{T'}.\mathit{MODE} \\
  \mathit{excp}_p^{T-t} &= \mathit{doutputs}_p^T.\mathit{excp}
  \end{aligned}$$

  Therefore, $\neg \mathit{doutputs}_p^T.\mathit{excp}$ holds.

- The processor runs in user mode, i.e. $dmode^T$ holds.

  First, we can show that $dmode^T = mode(c_S^{sI(m,T)})$. Due to the stability of the stage m during the processor request there is $sI(m,T) = sI(m,T')$ for the scheduling function. The processor request implies $\neg stallout^{T'-1}$ and $m^{T'}.valid$. From the computation of the flag $m^{T'}.valid$ (4.23) it follows that $RS_{MU}^{T'-1}.valid$. Therefore, the condition $dispatchRS_{MU}^{T'-1}$ holds by (4.41) and $sI(m,T') = sI(RS_{MU}, T'-1)$. Consequently, $sI(m,T) = sI(RS_{MU}, T'-1)$. Besides, $\neg stallout^{T'-1}$ implies $m^{T'}.MODE = RS_{MU}^{T'-1}.MODE$. Using $invRS_{MU}^{T'-1}$ it is easy to conclude

$$\begin{aligned} m^{T'}.MODE &= RS_{MU}^{T'-1}.MODE \\ &= mode(c_S^{sI(RS_{MU},T'-1)}) \\ &= mode(c_S^{sI(m,T)}) \end{aligned}$$

  Next, according to Lemmata 3.3.11 and 3.3.12 we have

$$\begin{aligned} & good\_proc\_interface(dtrc^t) \wedge \\ & good\_mem\_interface(dtrc^t) \wedge \\ & good\_tlb\_interface(dtrc^t) \implies \\ & \forall t', t'' \in \mathbb{N} : is\_req\_proc(t', t'', dtrc^t) \wedge \\ & \qquad mem\_ack\_pte(t', t'', dtrc^t) \wedge \\ & \qquad req_p^{t'} \wedge mode_p^{t'} \implies excp_p^{t''} = tr(t').excp \end{aligned}$$

  The premises are proven in the same way as in the previous case but with an additional Lemma 4.3.9. Hence, to use the assertion above we have to prove only $mem\_ack\_pte(T'-t, T-t, dtrc^t)$, that is for any cycles $t', t'' \in \mathbb{N}$ of the local $DMMU$ trace the following holds:

$$\begin{aligned} & t' < t'' \wedge t'' \in \left[T'-t : T-t\right] \wedge \\ & req_p^{t'} \wedge (\forall t \in \left[t' : t''\right] \ : \ mode_p^t) \wedge \\ & \left\langle addr_p^{t''}[28:9]\right\rangle \leq \left\langle ptl_p^{t''}\right\rangle \implies \\ & (ptea_p^{t''}[0] \implies mem^{t''}[ptea_p^{t''}[29:1]][63:32] = \\ & \qquad\qquad mem^{t'}[ptea_p^{t''}[29:1]][63:32]) \wedge \\ & (\neg ptea_p^{t''}[0] \implies mem^{t''}[ptea_p^{t''}[29:1]][31:0] = \\ & \qquad\qquad mem^{t'}[ptea_p^{t''}[29:1]][31:0]) \end{aligned}$$

  Since we consider the specification without interrupts we can prove the following:

– provided the processor is in user mode, the mode is not changed for all next instructions, i.e.

$$\forall i \in \mathbb{N}^+ : mode(c_S^i) \implies \forall j \in \mathbb{Z}_{\geq i} : mode(c_S^i) \tag{4.48}$$

– the page table origin and length are not changed in user mode, formally

$$\begin{aligned}\forall i \in \mathbb{N}^+ : mode(c_S^i) \implies& \\ \forall j \in \mathbb{Z}_{\geq i} : pto(c_S^i) = pto(c_S^j) \wedge& \\ ptl(c_S^i) = ptl(c_S^j)&\end{aligned} \tag{4.49}$$

Using the software condition *stablePT* and these two statements it is easy to show that the page table remains unchanged in user mode, namely

$$\begin{aligned}\forall i \in \mathbb{N}^+ : mode(c_S^i) \implies& \\ \forall j \in \mathbb{Z}_{\geq i}, ad \in PT(c_S^i) : c_S^i.M[ad[31:3]] =& \\ c_S^j.M[ad[31:3]]&\end{aligned} \tag{4.50}$$

Since $req_p^{t'}$ holds and the *DMMU* trace is computed so that $m^{t+t'}.valid$ is set, we can find a cycle before $t+t'$ when $m.MODE$ was updated and then remained unchanged. Therefore, the invariant for RS_{MU} allow us to conclude

$$m^{t+t'}.MODE = mode(c_S^{sI(m,t+t')})$$

So, we can consider the statements above for $i = sI(m, t+t')$.

Turning back to the assertion to be proven one sees that $ptea_p^{t''}$ does not point outside the page table as indicated by the premise $\left\langle addr_p^{t''}[28:9]\right\rangle \leq \left\langle ptl_p^{t''}\right\rangle$, where

$$ptl_p^{t''} = m^{T'}.PTL = RS_{MU}^{T'-1}.PTL$$

Moreover, $ptea_p^{t''}$ is computed on basis of $RS_{MU}^{T'-1}.PTO$. We easily get

$$\begin{aligned}RS_{MU}^{T'-1}.PTO &= pto(c_S^{sI(m,T')}) \\ RS_{MU}^{T'-1}.PTL &= ptl(c_S^{sI(m,T')})\end{aligned}$$

Since $sI(m, t+t') \leq sI(m, T')$ (as an increasing function on time), the values of *pto* and *ptl* for these instructions are the same by (4.49). Therefore, $ptea_p^{t''} \circ 0^2 \in PT(c_S^{sI(m,t+t')})$.

Let us examine the memory configurations $mem^{t'}$ and $mem^{t''}$. Using the trace computation and the lemma's premise concerning the memory we argue $mem^{t''} = c_I^{t+t''}.M = c_S^{sI(M,t+t'')}.M$. During the processor request the value of the scheduling function for the visible memory is unchanged, i.e. $sI(M, t + t'') = sI(M, T)$. Besides, for the cycle T when the instruction commits we know $c_S^{sI(M,T)}.M = c_S^{sI(m,T)}.M$. Therefore, we get

$$mem^{t''} = c_S^{sI(m,T)}.M$$

Analogously, there is $mem^{t'} = c_I^{t+t'}.M = c_S^{sI(M,t+t')}.M$. Since $dreq^{t+t'}$ and $m^{t+t'}.valid$ are active, one can find a cycle $T''' \in [t + t' : T]$ when the instruction commits. So, the equality below holds:

$$mem^{t'} = c_S^{sI(m,T''')}.M$$

Since $sI(m, t + t') \le sI(m, T''')$, by (4.50) the following is true:

$$\begin{aligned}\forall ad \in PT(c_S^{sI(m,t+t')}) : c_S^{sI(m,T''')}.M[ad[31:3]] = \\ c_S^{sI(m,T)}.M[ad[31:3]]\end{aligned}$$

This proves $mem\_ack\_pte(T' - t, T - t, dtrc^t)$ as it is required.

Now, we deal with

$$\begin{aligned} doutputs_p^T.excp &= excp_p^{T-t} \\ &= tr(T' - t).excp \end{aligned}$$

In the same manner as it is done for $dmode^T = mode(c_S^{sI(m,T)})$ we reason:

$$\begin{aligned} m^{T'}.PTL &= ptl(c_S^{sI(m,T')}) \\ mr^{T'} &= mr?(c_S^{sI(m,T')}) \\ mw^{T'} &= mw?(c_S^{sI(m,T')}) \end{aligned}$$

and by Lemma 4.3.19

$$\begin{aligned} m^{T'}.ea &= ea(c_S^{sI(m,T')}) \\ m^{T'}.ptea &= dptea(c_S^{sI(m,T')}) \end{aligned}$$

Moreover, for the memory one explores $c_I^{T'}.M = c_S^{sI(m,T')}.M$ by the lemma's premise, the scheduling function stability, and Lemma 4.3.22.

Hence, we easily conclude

$$tr(T' - t).excp = dDecodeITr(c_S^{sI(m,T')}).excp$$

Because of $sI(m, T') = sI(m, T)$ we finish the proof of the lemma. □

A similar lemma concerns the address provided by the *DMMU* for the memory.

Lemma 4.3.24 *The memory address is correct in the MU at a cycle $T \in \mathbb{N}$ when a memory instruction being executed commits. Formally, the following holds:*

$$\begin{aligned}
&\mathit{initMU}(c_I^0.\mathit{MU}) \wedge (\forall t \in \mathbb{Z}_T : \neg \mathit{reset}^t \wedge \neg \mathit{JISR}(c_I^t)) \wedge \\
&(\forall t \in \mathbb{Z}_T : \mathit{invRS}_{\mathit{MU}}(t)) \wedge (\forall t \in \mathbb{Z}_{\leq T} : c_I^t.M = c_S^{sI(M,t)}.M) \wedge \\
&\mathit{commits}(T, sI(m,T)) \wedge \neg \mathit{doutputs}_p^T.\mathit{excp} \implies \\
&\mathit{data}^T.\mathit{addr} = \begin{cases} \mathit{dpa}(c_S^{sI(m,T)})[31:3] & \mathit{dmode}^T \\ \mathit{ea}(c_S^{sI(m,T)})[31:3] & \mathit{otherwise} \end{cases}
\end{aligned}$$

Using Lemmata 3.3.13 and 3.3.14 this lemma can be proven in the same manner as the previous one.

Now we can consider a lemma, which shows that the memory $c_I.M$ is updated correctly for a case without interrupts.

Lemma 4.3.25 *Provided no reset and interrupt occur before a cycle $T \in \mathbb{N}$, the memory content in the implementation till the cycle T corresponds to its specification, namely*

$$\begin{aligned}
&\mathit{initMU}(c_I^0.\mathit{MU}) \wedge (\forall t \in \mathbb{Z}_T : \neg \mathit{reset}^t \wedge \neg \mathit{JISR}(c_I^t)) \wedge \\
&(\forall t \in \mathbb{Z}_T : \mathit{invRS}_{\mathit{MU}}(t)) \implies (\forall t \in \mathbb{Z}_{\leq T} : c_I^t.M = c_S^{sI(M,t)}.M)
\end{aligned}$$

Proof: We prove this lemma by induction on T.
Induction base ($\mathbf{T = 0}$): For this case the memory both in the specification and in the implementation corresponds to its initial state *init_mem*. This finishes the case.
Induction step ($\mathbf{T \to T+1}$): For the induction step we are to prove that the following holds:

$$\forall t \in \mathbb{Z}_{\leq T+1} : c_I^t.M = c_S^{sI(M,t)}.M$$

Owing to the induction hypothesis the only claim to be considered is

$$c_I^{T+1}.M = c_S^{sI(M,T+1)}.M$$

We split cases depending on $sI(M,T)$ update.

- No instruction finishes its memory access: $(\neg mr^T \wedge \neg mw^T) \vee \mathit{dbusy}^T$.

  Therefore, the value of the scheduling function is not changed, and $sI(M, T+1) = sI(M,T)$. Using the induction hypothesis we conclude $c_S^{sI(M,T+1)}.M = c_I^T.M$.

  Let us consider both the parts of the disjunction above.

– If $\neg mr^T \wedge \neg mw^T$ holds, it easy to show $\neg data.mw^T$, and in turn $c_I^{T+1}.M = c_I^T.M$. So, the case is proven.
– If $dbusy^T$ holds, then from the *DMMU* design we conclude

$$\neg(doutputs_m^T.mw \wedge \neg dinputs_m^T.busy)$$

Therefore, the computation of the signals in *DPFCTRL* implies

$$\neg(data^T.mw \wedge \neg data^T.busy)$$

and the memory in the implementation is not updated.

- An instruction finishes its memory access: $(mr^T \vee mw^T) \wedge \neg dbusy^T$.

  We consider cases depending on mw^T.

  – Let $\neg mw^T$ hold. Obviously, the memory in the implementation is not updated, namely $c_I^{T+1}.M = c_I^T.M$. Due to the fact that the instruction commits, we get $c_S^{sI(M,T+1)}.M = c_S^{sI(m,T)+1}.M$.
  Since there is a read instruction in the *MU*, one can prove $\neg mw?(c_S^{sI(m,T)})$. Therefore, the memory remains unchanged and $c_S^{sI(m,T)+1}.M = c_S^{sI(m,T)}.M$ holds. Using the induction hypothesis and Lemma 4.3.22 we finish the proof of this case.
  – Let mw^T hold. Using Lemma 4.3.20, it is easy to prove that $\neg dmal(c_S^{sI(m,T)})$ holds.
  If this is a request to the external devices, the memory $c_I^T.M$ is not updated and the proof runs in the same way as in the previous case.
  Otherwise, the proof depends on the flag $dpf(c_S^{sI(m,T)})$. By Lemmata 4.3.23 and 4.3.24 we can state

  $$dexcp^T = dpf(c_S^{sI(m,T)})$$

  Now, we consider the cases:
  * If the flag $dpf(c_S^{sI(m,T)})$ is set, the proof is similar to the case with $\neg mw^T$.
  * Otherwise, we consider how the memory is updated. Since the stage m contains the correct information for the instruction $sI(m,T)$ (the correctness of $shift4store$ can be found in [Bey05]), using Lemmata 3.3.10 and 3.3.12 we make sure that the proper data is written into the memory correctly. This finishes the proof of the lemma.

□

Beside the memory update performed by the memory unit, we have to cover the second point of the *MU* correctness. So, we provide the lemma concerning the *MU*'s outputs.

Lemma 4.3.26 *Provided no reset and interrupt occur before a cycle $T \in \mathbb{N}$ when the results of the memory instruction execution are ready, the outputs of the MU at this cycle are correct, namely*

$$\begin{aligned}
&initMU(c_I^0.MU) \wedge (\forall t \in \mathbb{Z}_T : \neg reset^t \wedge \neg JISR(c_I^t)) \wedge \\
&(\forall t \in \mathbb{Z}_T : invRS_{MU}(t)) \wedge valid_{P_{MU}}^T \implies \\
&data_{P_{MU}}^T = data(c_S^{sI(m,T)}) \wedge \\
&dmal_{P_{MU}}^T = dmal(c_S^{sI(m,T)}) \wedge \\
&dpf_{P_{MU}}^T = dpf(c_S^{sI(m,T)})
\end{aligned}$$

We omit the proof because it proceeds in a similar way as for the lemmata above. All the lemmata considered before in this section can be used in this proof.

As for the correctness with interrupts we are only interested in the memory update. So, we have to prove that if the first interrupt $JISR(\tilde{c}_I^T)$ occurs at a cycle $T \in \mathbb{N}$, then the equality $c_I^{T+1}.M = \tilde{c}_S^{sI(wb,T+1)}.M$ is valid.

4.3.3 The MU Correctness for Instruction Fetch

In the pipelined processors we deal with a self-modifying code because an instruction just has been fetched from the memory could be overwritten by a store instruction being executed in the memory unit. To tackle such RAW hazards in the VAMP there is used a so-called `sync` instruction [Bey05,Dal06] before fetching from a modified position in the memory. The `sync` instruction drains the pipeline of the processor.

The correctness of the instruction fetch for the VAMP is covered in [Dal06, p. 74–86] on condition that the program code is synced. Here, we only state a lemma without proofs for the instruction fetch in the memory unit described in this work.

Lemma 4.3.27 *Provided no reset and interrupt occur before a cycle $T \in \mathbb{N}$ and the sync condition on the assembler code hold, the instruction fetch is performed correctly till this cycle. Formally, we have*

$$\begin{aligned}
&init(c_I^0) \wedge (\forall t \in \mathbb{Z}_T : \neg reset^t \wedge \neg JISR(c_I^t)) \implies \\
&(\forall t \in \mathbb{Z}_{\leq T} : c_I^t.S1.full \implies c_I^t.S1.IR = IR(c_S^{sI(issue,t)}) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad c_I^t.S1.imal = imal(c_S^{sI(issue,t)}) \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad c_I^t.S1.ipf = ipf(c_S^{sI(issue,t)}))
\end{aligned}$$

Chapter 5

Conclusion

5.1 Summary

It this thesis we presented the implementation of the memory management unit with the translation look-aside buffer for the fast address translation and formally verified the correctness of the *MMU*. The *MMU*'s design covered in this work is based on the construction introduced in [Dal06] but contains extensions proposed there.

- Because of using the *TLB* now the *MMU* is able to translate the virtual address without accessing the physical memory if the buffer contains an appropriate page table entry. This greatly decreases the latency of *MMU* requests.

- Another extension allows to distinguish between memory pages containing data and the executed code. The *MMU* prohibits the instruction fetch from the data region in the memory.

- The *MMU* is also optimized by taking the computation of the page table entry address out the memory management unit so that the *MMU*'s latency is additionally reduced by one cycle in comparison with the *MMU* covered in [Dal06].

As a next step, we integrated this new *MMU* into the memory unit of the VAMP processor considered in [Dal06]. This required to develop a circuit computing the effective and page table addresses simultaneously. For the instruction fetch we also provided a trivial *ptea* computation circuit. Along with these changes we made another couple of extensions for the *MU*.

- The extended design of the memory unit supports an interface for the external devices [Tve08]. Any access to the external devices must not be cached. So, the *MU* communicates directly with a device via the synchronous bus protocol [MP00, Bey05] in case of a single word load/store instruction.

- The *MU* allows not to stall the execution of a store instruction while it is not the oldest instruction in the pipeline. For such an instruction the request to the *DMMU* is generated and only a write request to the cache system or an external device is stalled. So, the read access for the page table entry can be started earlier in contrast to [Dal06].

For these extensions we developed universal for both the *MMU*s *prefetch control logic* and an additional logic for generating a request to a device on basis of the *DMMU* output signals and detecting that the device is busy.

We also formally specified the memory unit in the context of the programmer's model of the VAMP and showed the correctness criteria. Note that we did not focus on accessing the external devices since that is a matter of [Tve08]. In the proofs we paid attention to the memory update operation. Though the proof of the overall *MU* correctness is not complete, it is seen how to use the proofs of the *MMU* for the *MU* verification.

5.2 Future Work

The future work should be done in two directions: the formal verification and the *MU* design extensions.

We need to finish the verification of the *MU* overall correctness. This is a large amount of work due to a few reasons. Since this thesis is based on the formal verification of the VAMP carried out in PVS, all the previous proofs do not match the interactive theorem prover Isabelle/HOL and cannot be used without translation. One more reason is the substantial modifications made in this work for the previous *MU* design.

Most of the proofs in this thesis were done interactively in spite of coupling Isabelle with external model checkers NuSMV and SMV. Actually, the user's involvement could be considerably reduced as reported in [Tve08]. Therefore, we should adapt the *MU* verification for automated methods.

The optimization and extensions were applied to the *MMU* and the memory unit in whole in this work to make them satisfy the requirements for the industrial processors. However, the are still possible ways for the further development.

- So, the multi-level address translation was proposed in [Hil05] as a more space efficient extension. Now it could be realized at the next step of the VAMP virtual memory support development.

- Pipelining the VAMP memory unit is also reasonable because the memory accesses in user mode are performed in two steps. With this modification the *MU* could process more than one instruction at a time.

- One more possible extension is address space protection for processes sharing the memory.

Appendix A

Mapping to Lemmata in Isabelle/HOL

In this appendix we give a mapping from the lemmata and theorems considered in this thesis to corresponding proofs in Isabelle/HOL.

Number	Thesis Page	Name in Isabelle
3.1.18	25	mem_live_impl_mem_live_strong
3.3.1	32	mmu_local_correct_thm
3.3.2	32	proc_inputs_stable_for_req
3.3.3	33	mem_inputs_stable_for_req
3.3.4	33	mem_mr_mw_mutex_lem
3.3.5	33	mem_mr_stable
3.3.6	34	mem_inputs_stable_lem
3.3.7	35	proc_liveness_lem
3.3.9	37	untranslated_read_lem
3.3.10	38	untranslated_write_lem
3.3.11	40	translated_read_lem
3.3.12	47	translated_write_lem
3.3.13	48	untranslated_address_correct
3.3.14	48	translated_address_correct

Table A.1: Lemmata for the local *MMU* correctness

Number	Thesis Page	Name in Isabelle
4.3.7	74	dmmu_proc_mr_mw_fetch_mutex
4.3.8	74	dmmu_proc_inputs_stable
4.3.10	75	dtlb_data_consist
4.3.11	75	dtlb_purge_comp
4.3.15	76	commits_imp_req_proc_data
4.3.16	76	commits_imp_req_proc_data_helper
4.3.17	78	rollback_valid_mutex
4.3.18	79	is_req_proc_dmmu_imp_is_req_proc
4.3.19	79	EA_PTEA_correct
4.3.20	80	mu_dmal_correct_for_req_proc
4.3.21	80	init_state_for_trace_exists
4.3.22	81	mem_conf_const_helper

Table A.2: Lemmata for the *MU* correctness

Bibliography

[ASG04] P. B.Galvin A. Silberschatz and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 7th ed. edition, 2004.

[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.

[BJ01] Christoph Berg and Christian Jacobi. Formal verification of the VAMP floating point unit. In *CHARME 2001*, volume 2144 of *LNCS*, pages 325–339. Springer, 2001.

[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.

[CCG$^+$02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[Dal06] Iakov Dalinger. *Formal verification of a processor with memory management units*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[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.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctnesss.* PhD thesis, Saarland University, Saarbrücken, Germany, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[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.

[JM98] Bruce Jacob and Trevor Mudge. Virtual memory: issues of implementation. *Computer*, 31(6):33–43, 1998.

[Krö01] Daniel Kröning. *Formal Verification of Pipelined Microprocessors.* PhD thesis, Saarland University, Saarbrücken, Germany, 2001.

[McM99] K. L. McMillan. *Getting started with SMV.* Cadence Berkeley Labs, 1999.

[Mül07] Christian Müller. Verification of a simple cache system. Master's thesis, Saarland University, Saarbrücken, Germany, 2007.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness.* Springer, 2000.

[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.

[Tve05] 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.

[Tve08] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems (Draft).* PhD thesis, Saarland University, Saarbrücken, Germany, 2008.

[Ver03] The Verisoft Project. http://www.verisoft.de/, 2003.