Correctness of Multi-Core Processors with Operating System Support

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurswissenschaften
der Fakultät für Mathematik und Informatik
an der Universität des Saarlandes

eingereicht von
Petro Lutsyk

Saarbrücken, Oktober 2018

Tag des Kolloquiums:	27. September 2018
Dekan:	Prof. Dr. Sebastian Hack
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Verena Wolf
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Dr. Silvia Melitta Müller
Akademischer Mitarbeiter:	Dr. Jonas Oberhauser

Zusammenfassung

Im Zuge der Unterstützung von Hypervisoren verifizieren wir einen realistischen Pipeline-Multi-Core-Prozessor mit integriertem Mechanismus für die zweiphasige (verschachtelte) Adressübersetzung. Das verschachtelte Übersetzungsschema wird benötigt, damit Gäste des Hypervisors (typischerweise Betriebssysteme) ihre Programme im übersetzten Modus ausführen können. Wir betrachten das Setup, in dem die Betriebssysteme als Prozesse (im übersetzten Modus) des Hypervisors laufen, 'auf der bloßen Hardware', d.h. ohne Adressübersetzung.

Die geschachtelte Übersetzung wird von der geschachtelten Memory Management Unit (MMU) durchgeführt, wobei beide Übersetzungsphasen in Hardware ausgeführt werden. Sowohl die Spezifikation als auch die Implementierung der geschachtelten MMU werden ausführlich dargestellt. Es wird bewiesen, dass die geschachtelte MMU eine allgemeinere Hilfsspezifikation, die die geschachtelte MMU vom Rest der Maschine isoliert, korrekt implementiert. Letzteres erlaubt es uns, die Argumente für die Korrektheit der MMU-Implementierung in jeder Maschine auf eine einfache Simulation zwischen zwei Softwaremodellen zu reduzieren.

Der Hauptbeitrag dieser Arbeit ist der vollständige Korrektheitsbeweis auf Papier für die Pipeline-Multi-Core-Implementierung des MIPS-86 ISA, der zur Unterstützung der verschachtelten Übersetzung zusätzlich erweitert wurde. Wie der Name schon vermuten lässt, kombiniert MIPS-86 den Befehlssatz von MIPS mit dem Speichermodell von x86. Zuerst betrachten wir diese erweiterte MIPS-86-Spezifikation in der sequentiellen Implementierung, die dazu dient, die Integration der verschachtelten MMUs in den MIPS-Prozessor zu demonstrieren und der Einfachheit halber auf einen einzelnen Prozessorkern beschränkt ist. Im Beweis unseres Hauptergebnisses — Korrektheit der Pipeline-Implementierung — verweisen wir auf den sequentiellen Fall, um die Korrektheit der MMU-Operation zu zeigen. Dies erlaubt uns, den Fokus auf die Probleme des Pipelinings der Maschine mit spekulativer Ausführung und Unterbrechungen zu verlagern, die bei vorhandener Adressübersetzung zu berücksichtigen sind.

Abstract

In the course of adding support for hypervisors, we verify a realistic pipelined multi-core processor with integrated mechanism for two-phase (nested) address translation. The nested translation scheme is required to allow guests of the hypervisor (typically operating systems) to execute their programs in translated mode. We consider the setup in which the operating systems are running as processes (in translated mode) of the hypervisor, running 'on the bare hardware', i.e., without address translation.

The nested translation is performed by the nested memory management unit (MMU), with both phases of translation performed in hardware. Both the specification and the implementation of the nested MMU are presented in full detail. The nested MMU is proven to correctly implement an auxiliary, more general specification which isolates the nested MMU from the rest of the machine. The latter allows us to reduce arguments on correctness of the MMU implementation in any machine to a simple simulation between a pair of software models.

The main contribution of this thesis is the complete paper and pencil correctness proof for the pipelined multi-core implementation of the MIPS-86 ISA, additionally extended to support the nested translation. As the name suggests, MIPS-86 combines the instruction set of MIPS with the memory model of x86. First, we consider this extended MIPS-86 specification in the sequential implementation, which serves to demonstrate integration of the nested MMUs into the MIPS processor and for simplicity is restricted to have a single processor core. In the proof of our main result — correctness of the pipelined implementation — we refer to the sequential case to show correctness of the MMU operation. This allows us to shift the focus towards the problems of pipelining the machine with speculative execution and interrupts, which are necessary to consider in the presence of address translation.

Acknowledgments

First and foremost, I would like to thank Prof. Dr. Wolfgang J. Paul for an opportunity to study computer architecture at his chair and write a doctoral thesis under his supervision. Starting from the very first lecture, about six years ago, it is hard to remember a meeting where he would not try to share some of his experience, no matter that was in a lecture hall, on a dance floor, or in his kitchen. I thank Prof. Paul for his catching enthusiasm, endless optimism and patience. Needless to mention that most of the mathematical culture necessary to write this dissertation I learned from him, both through lectures and private lessons.

Also, I want to thank my colleagues, ever working at the chair, for keeping a friendly and creative atmosphere. Especially, I want to thank my friend Dr. Jonas Oberhauser for being always ready to answer (many of) my questions and help with an advice, being a responsive room mate and a great friend. Also, I am grateful to Jonas for giving me a hard time in our scientific discussions, which after all only motivated me to keep learning.

Finally, I want to thank my family and friends, who kept me motivated to complete this dissertation. First, I thank my parents, who always believed in me and gave me a chance to travel to Germany and try myself here. Also, I am thankful to my brother and his family, who took care of me during my stay in Germany from the first day. Last but not least, I thank my first, and hopefully last, dancing partner for her support and patience during the whole time together.

Contents

List of Figures

List of Tables

Introduction

The last decade witnessed a revolution at the market of mobile computing, blurring the boundary between ordinary computers and various specialized electronic devices. Today everything, from mobile phones to toothbrushes, is capable of network communication. Mobile phones, for instance, are no longer supplied with simple chips, providing some limited, highly specialized set of features. Now they are more similar to ordinary computers. In fact, they are computers, computers with multi-core processors inside [ARM14]. Differences between today's desktop PC, laptop, tablet, and mobile phone are rather quantitative than qualitative, and determine *only* the performance of hardware. Leaving the performance aspects aside, all these devices, running similar programs within similar operating systems, execute equivalent code. Besides the obvious benefits for uses and software developers, such unification of hardware platforms creates advantages for hackers and developers of malicious software [ELMC18].

The tendency to equip daily life devices with microprocessors is expected to grow over time. This also concerns life-critical systems driving airplanes, trains, cars, etc. Programming such systems requires a comprehensive understanding of operational behavior of integrated processors. Operational behavior is normally described in manuals, which for modern processors are typically several thousand pages long [Int16]. Fortunately, industrial engineers simply do not trust things they do not understand. In this regard, for many years they avoid using many of the features provided by modern processors, including parallelism of multi-core CPUs. Consequently, the total number of processors and microcontrollers installed, for instance, in a single vehicle is reaching one hundred. This is a true problem for the industry with sizeable risks for the customers: in a highly competitive environment, such as the automotive industry, it becomes a matter of time when dozens of chips will be replaced by a handful of powerful multi-core processors. Solving this problem and mitigation of the risks becomes one of the central tasks in modern computing.

One possible solution is provided by formal verification of computer systems. As depicted in Fig. 1, computer systems are formally described by models which justify computations on various system layers: from the physical layer at the bottom — via the (OS) kernel layer — to the applications at the top. The following principle was formulated almost thirty years ago (see *Related Work*) but has not lost its relevance: correctness of the upper system layers hinges on correctness of the lower layers, and is derived by refining the models at the upper layers by the models at the lower (adjacent) layers. Clearly, in order to show correctness of the entire stack, one has to prove that every model is refined by the model below.

According to Fig. 1, as one should not try to prove the laws of nature, one should start at the layer of physical hardware. Having correct policies to abstract away dynamic RAMs and buses [KMP14], one can start at the layer of digital hardware. Of course, one can completely ignore the complexity of industrial processors and avoid reasoning about the real hardware: some of the verification projects exclude the hardware layers from the model stack, and consider only the software layers [SK17, KAK$^+$]. These approaches rely on certain properties, such as 'non-bypassable architectural constraints' in [KAK$^+$], guaranteed by most of the commercial chip manufacturers. As practice shows, such guarantees can be discovered broken after they were trusted for many years. For instance, security of modern computer systems heav-

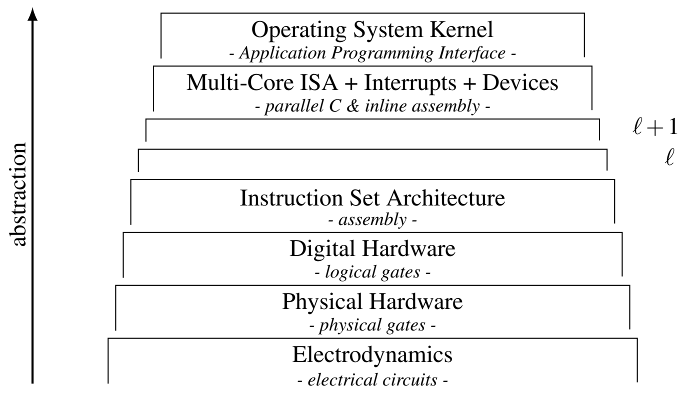


Fig. 1: Formal representation of computer systems.

The models describing a computer system altogether form a stack, in a sense that the model at layer ℓ implements the model at layer $\ell+1$. That is, to justify semantics of the model implemented at layer $\ell+1$, one has to prove that the model at layer ℓ *refines* the model at layer $\ell+1$, i.e., that for every (low-level) computation at layer ℓ there is an equivalent (high-level) computation at layer $\ell+1$. For instance, one cannot justify semantics of the instruction set architecture (ISA) without establishing its refinement by the digital hardware.

ily relies on isolation of memories (data) of programs running on the same physical machine. In [LSG⁺18] the authors demonstrated how to exploit side effects of out-of-order (speculative) execution to read the memory of the kernel; in [KGG⁺18] the memory of the victim's process. Since speculative execution has become an industrial standard decades ago [Int18], billions of devices from the main suppliers (including microprocessors from Intel, AMD, and ARM) were found vulnerable to the attacks reported in [LSG⁺18, KGG⁺18]. In order to avod similar consequences, one should start verification of computer systems (model stacks) at the layer of digital hardware.

Related Work

First attempts for formal verification of computer systems were made in the late 1980's, when an approach of *systems verification* was demonstrated to verify the model stack from Computational Logic, Inc. or CLI stack for short [BHMY89, Bev89]. The CLI stack consisted of the following components:

i) a 32-bit microprocessor (FM8502) designed (ISA specification), implemented at the gate-level and verified by Warren Hurt;
ii) assembly language (Piton) with the assembler, linker, and loader designed, implemented and verified by J Moore with help from Matt Kaufmann;
iii) two high-level languages with compilers implemented and verified by Bill Young (micro-Gypsy) and Art Flatau (subset of Nqthm Pure Lisp); and
iv) the operating system (KIT) designed, implemented and verified by Bill Bevier.

After formalizing the Netlist Description Language (NDL) in 1992, Bishop Brock and Warren Hurt designed (ISA) and verified a new microprocessor (FM9001) they described using NDL. The Piton assembly language then was ported to the ISA of FM9001 by J Moore, who updated and reverified the code generators. This allowed the rest of the CLI stack to be ported mechanically [Moo03].

In the 1990's, with the rapid development of computer systems, the CLI stack quickly fell short for many reasons. For instance, the FM9001 microprocessor implemented an unrealistically simple architecture: no pipeline, no speculation, no floating-point unit, no cache. Though the FM9001 implemented memory mapped I/O, none of the verified high-level languages had facilities to support it. Overall, both high-level languages were too simple to be of practical use. In the early 2000's, the results of the CLI stack project were revisited by one of its principle architects, and the 'grand challenge' for *pervasive verification* of computer systems was proposed to the formal methods community. The challenge was to design and mechanically verify a practical computer system, from gates to software [Moo03].

Within the time frame from 2003 to 2010 a new effort to formally verify model stacks was undertaken in the Verisoft and Verisoft XT projects [Ver07, Ver10]. Meeting the challenge of [Moo03], in the course of the Verisoft project the 'short' CLI stack was extended by integrating devices and targeting a more realistic system architecture regarding both hardware and software [DDB08, APST10]. However, an attempt to transfer the theory developed for sequential systems onto multi-core architectures made in the course of the Verisoft XT was less successful. The main reason: lack of a pervasive theory of multi-core systems [Sch13b].

More recently — about five years ago — the missing guidelines were outlined [CPS13]. Following these guidelines, at the layer of hardware — the scope of this thesis — the gate-level implementation of a multi-core CPU must be verified to implement its ISA specification. Since commercial hardware vendors tend not to disclose layouts of their products, that multi-core CPU first had to be designed and implemented. The resulting design included support for most of the principal features that the modern industrial processors provide: multiple physical cores, pipelining, multi-level address translation, internal and inter-processor interrupts, devices, etc. All these mechanisms were specified in the doctoral thesis of Sabine Schmaltz [Sch13b]. Progress on this work is described in detail below.

In 2014, laying down the groundwork for multi-core hardware verification, the gate-level implementation of a realistic RISC processor was proven correct [KMP14]. This was an implementation of a subset of the MIPS ISA, in particular containing full gate-level designs for both a pipelined processor with hardware interlock and a sequentially consistent cache memory system implementing the MOESI protocol. Afterwards, in a series of bachelor and master theses, that implementation was successfully

- synthesized on an FPGA board [Mai14] and
- tested (four-core version) by numerical computation programs [Ols14], and
- extended to support store buffers (single-core version) [Lut14].

The first attempt to extend the design from [KMP14] with mechanisms to support hypervisors (operating systems) was taken in the scope of the university lecture on multi-core system architecture at Saarland University in late 2015. The multi-level address translation scheme used in the lecture was previously designed and proven correct for a sequential single-core machine in [Sch14a]. Integration of inter-processor interrupts into a multi-core machine with sequential processors was covered in [Sch14b]. The memory management units (MMUs) and advanced programmable interrupt controllers (APICs), constructed in [Sch14a] and [Sch14b] resp., were integrated into the stages of the pipelined machine from [KMP14] and presented to students. The design was preliminarily extended to support internal interrupts. Throughout 2016 the ideas presented in the lecture were summarized in the early version of the lecture notes [Pau16].

Initially we believed that a full correctness proof for the new design, given the sketch in [Pau16], could be covered in a series of master theses. Correctness of the internal interrupt mechanism added in [Pau16] was first proven for a simpler design, namely for a five-stage pipeline from [KMP14], in [Sch16a]. Unfortunately, the approach used in [Sch16a] turned out to be not powerful enough to handle the additional pipeline stages added for address translation. In case of a rollback in the resulting seven-stage pipeline with speculative execution, one has to stabilize signals in up to three pipeline stages in which the memory system is accessed, but the approach used in [Sch16a] could only be applied to stabilize signals in one of the rolled-back stages. An improved mechanism which allows to stabilize signals in arbitrary number of rolled-back stages was developed and proven correct by Jonas Oberhauser in [LOP]. The obtained seven-stage pipelined machine (single-core version) with the rollback mechanism of [LOP] was successfully implemented on an FPGA board [Zah16], building on top of the practical realization of the five-stage pipeline from [Mai14].

Finally, correctness of the simplified inter-processor interrupt (IPI) mechanism was proven for a multi-core machine with sequential processors in [Sch16b]. There the IPI mechanism is restricted to support only the interrupts broadcast by the local APICs. Integration of the IPI mechanism as specified in [Sch13a] (with an I/O APIC and a device subsystem) into the machine from [LOP] is still, at the moment of writing, the subject for future work.

A different approach on formal verification of hardware is conducted by researches in the Massachusetts Institute of Technology. In contrast to the standard approach, to model processors as synchronous finite state machines (FSM), the authors of [VCAD15] utilize the labeled transition systems (LTS), a more abstract model of computation compared to FSM. In particular, in their work the authors show how to use the model to

i) decompose hardware systems s.t. each component is modeled by a separate LTS and composition of LTSes of all components models the whole system, and
ii) replace a component with its simper substitute *if* the LTS associated with that component implements the LTS associated with its substitute.

These techniques allow the authors to formally verify the LTSes representing fairly complex designs, including speculative processors and hierarchical cache memory systems. On the other hand, usage of a more abstract model has certain drawbacks: while the model of FSM permits a straightforward implementation at the register transfer level (RTL), the same is not obvious for the model of LTS. To synthesize their design, the authors of [VCAD15] substitute the LTS describing a processor by a program in a dialect (BlueSpec) of a high-level functional language (Haskell); the latter representation is proven isomorphic to the one using LTSes. Therefore, the proposed approach hinges on correctness of the proprietary compiler, which translates BlueSpec descriptions to RTL (System Verilog). In the open literature we managed to find the results on correctness of a compiler for a rather academic, simplified version of BlueSpec (Fe-Si) [BC13]. Correctness of the BlueSpec compiler still, up to our knowledge, remains an open question [Vij16, CVS+17]. Of course, that is insufficient to guarantee correctness of the entire model stack as described in [Moo03].

Finally, we cannot ignore the results reported in [CVS+17]. There the authors present a framework (Kami) for the Coq proof assistant, which allows to i) design and verify "fairly realistic processors" in Coq, and then ii) extract designs synthesizable on an FPGA board. The Kami framework uses the BlueSpec compiler as a subroutine. As a case study, the authors of [CVS+17] verify a multi-core processor, consisting of pipelined (four stage) cores and a sequentially consistent cache memory system. In addition, the cores are equipped with branch predictors, but do not implement any forwarding mechanisms. The model was instantiated with the RISC-V [RIS18] cores implementing the subset of RV32I, the instruction set consisting of 32-bit integer portion of the open-source RISC-V ISA [WLPA16], without subword memory accesses. The design was synthesized to have the four-core processor and the memory system with direct-mapped caches. The processor produced by the Kami framework was reported to be 4% *slower* (per core) compared to the sequential implementation of the entire RV32I produced by the BlueSpec compiler from the design in [WZB+16].

In our opinion, the results above look somewhat strange. In [MP00] performance of the sequential implementation of a simple DLX processor was compared against its pipelined version, also featuring no forwarding. At the moment of writing, the latter results were published almost twenty years ago. According to the presented analysis, the speedup gained by pipelining normally reaches 120%, provided that a low-latency (cache) memory system is used.

Goals and Approach

Hypervisors are kernels that allow to run multiple operating systems as processes on a single physical machine [ABCC66]. If one wants to run user programs within these operating systems (which is the standard scenario), an additional phase of address translation is required. The multi-level address translation scheme utilized in [Pau16] provides only one phase of translation. While there are software techniques to deploy the second phase of translation (actually, any number of translation phases), these techniques put an additional load on the underlying hypervisor [Meg12, Kov13]. In order to avoid the aforementioned overheads, we replace the translation scheme of [Pau16] with a more powerful one, which provides support

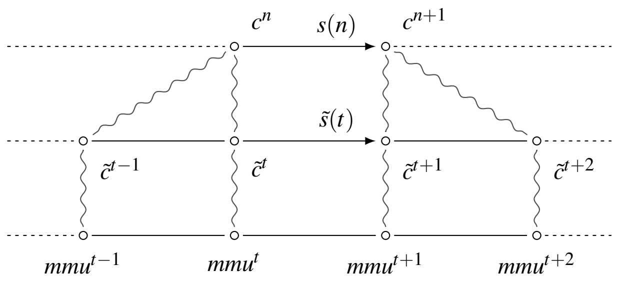


Fig. 2: Two-step simulation of the MMU computation.

At the bottom: hardware computation (mmu^t); in the middle: general computation ($\tilde{c}^t$); at the top: ISA computation (c^n). TLB steps are generated by the hardware computation in cycle t. Oracle input $s(n)$ for the step performed in the semantics of [Sch13a] is provided by stepping function s, whereas oracle input $\tilde{s}(t)$ for the step performed in the general semantics (Sect. 3.4) is provided by stepping function $\tilde{s}$. Simulation relation between configurations of hardware mmu^{t+1} and ISA c^{n+1} is established in two steps: first by showing simulation (sim_{tlb}) between mmu^{t+1} and general configuration $\tilde{c}^{t+1}$ (Chap. 5), and then by showing simulation (sim_{tlb}^{ISA}) between (software) configurations $\tilde{c}^{t+1}$ and c^{n+1}.

for two phases of address translation in hardware. Therefore, the main goal of this thesis is to strengthen the virtualization capabilities of the machine from [Pau16] by designing the two-phase scheme for (nested) address translation (NAT). Certainly, we are to prove correctness of the resulting design in the spirit of [KMP14].

Usually integration of new mechanisms like store buffers or multi-level address translation into the basic design from [KMP14] did not require significant changes of specifications or modifications of constructions already proven correct. The necessary proof efforts in most cases were the subject for bachelor or master theses like [Lut14], [Sch14a], [Sch16b]. With NAT things turn out to be slightly more involved. Since the scheme of NAT is inherently quite complex, it requires considerably more efforts to specify, construct, and integrate into the basic design. Moreover, since NAT is used to translate both the instruction address (program counters) and the effective addresses (memory accesses), a necessity to extract the correctness of NAT from the arguments on overall machine correctness arises naturally.

Therefore, we handle NAT as follows. Immediately after presenting its formal specification we introduce another, more general specification of address translation. The general specification considers a single configuration component — the translation look-aside buffer (TLB), representing a set of translations — and defines the semantics by specifying how the translations can be added to or dropped from this set. We show this specification to be equivalent to the specification from [Sch13a] in case i) only the TLB component is considered and ii) a number of conditions are fulfilled. After we design the memory management unit (MMU), we prove that it correctly implements the general specification. This allows us to separate correctness of the MMU implementation from the other correctness arguments (see Fig. 2). Moreover, this proof is completely independent of the machine type in which the MMU components are utilized.

Thus, in one of the chapters (Chap. 7) we show how to integrate NAT into a simple (single-core, sequential) machine. There we can reveal important proof goals without blurring the arguments related to the MMU correctness by the machine-specific details. The reader can treat this chapter as a repetition before doing the main proof, just like a training session before going into the wild. Afterwards, we reuse many results obtained in this chapter literally, replacing only the machine-specific lemmas involved. It becomes obvious that integration of NAT boils down to justification of conditions formulated together with the general specification, under which the two specifications of TLB are equivalent (i.e., showing sim_{tlb}^{ISA} as explained in Fig. 2).

That not only makes our argument more modular, but also streamlines the correctness proof. Modularity comes at the price that things become more lengthy overall and some mathematically redundant arguments are introduced. This mostly concerns parts where we interface the components and therefore the proofs. We pay this price in favor of clarity and flexibility s.t. the obtained results could be used in the future.

Outline

The material in this thesis is arranged in four major parts. In the first part we present all basic definitions and notations used throughout the entire text (Chap. 1). For convenience, in the first part we also include specifications of most of the components we build on (Chap. 2).

Part two is entirely devoted to nested address translation. Chapter 3 presents the process of NAT, both informally and formally, by specifying the MIPS ISA with NAT. In this chapter we also specify the new basic hardware mechanisms which we integrate: the privilege levels (levels of execution) and the intercepts. In the next chapter (Chap. 4) we implement a processor component which performs NAT in hardware. This component we call the nested memory management unit or the nested MMU for short. Correctness and liveness of the nested MMU are both proven in the end of part two (Chap. 5) of the thesis.

In the third part we consider the nested MMU in the sequential single-core implementation of MIPS. The sequential case is included in order to present the important proof goals on a simpler hardware. Thus, in Chap. 6 we interconnect a sequential core, two nested MMUs, and four caches of the memory system from [KMP14]. Next, in Chap. 7 we show how to integrate the correctness results from Chap. 5 to establish correctness for the TLB component.

Finally, in part four we consider the nested MMU in the (modified) pipelined multi-core implementation from [KMP14]. First, in Chap. 8 we interconnect a pipelined core, two nested MMUs, and four caches of the memory system from [KMP14], exactly as we connected in part three. Then, in Chap. 9 we consider multiple processors from Chap. 8 connected to a single cache memory system in parallel, where each processor is connected to four (private) caches. The arguments from Chap. 7 are very helpful in Chap. 9, which covers most of the new correctness results presented in this thesis. Using these arguments allows us to streamline the presentation in Chap. 9 and focus on problems related to pipelined implementation.

Part I

Single-Core MIPS with Address Translation

1 Definitions and Notation

This is actually Sect. 1.3 of [LOP]; it collects some basic definitions and notation we use massively throughout the entire text. Sections 1.1–1.3 summarize the material from Chap. 2 of [KMP14]. Section 1.4 describes the memories used in software and hardware models; also it contains the definition of access based memory semantics from Chap. 8 of [KMP14]. All these materials are included exclusively for completeness of presentation. The authorship over these materials belongs to all the authors of [LOP].

1.1 Sets, Sequences, and Records

We denote by

$$\mathbb{N} = \{0,1,2,\ldots\}$$

the set of natural numbers including zero by

$$\mathbb{Z} = \{\ldots,-2,-1,0,1,2,\ldots\}$$

the set of integers and by

$$\mathbb{B} = \{0,1\}$$

the set of Boolean values. For $i < j$ intervals of integers are defined as

$$[i:j] = \{i, i+1, \ldots, j\}.$$

In mathematics, finite or infinite sequences a of elements a_i are usually indexed starting from 1, i.e., they are written as

$$a = (a_1, \ldots, a_n)$$

resp.

$$a = (a_1, a_2, \ldots).$$

For computations or their sequences of inputs and outputs it is more convenient to start indexing with 0, i.e., to write

$$(c^0, c^1, \ldots).$$

In this way c^0 is the start configuration and c^i is the configuration reached after i steps. Finally for finite bit strings b it is most convenient to number sequences from right to left starting with 0:

$$b = (b_{n-1}, \ldots, b_0).$$

For finite subsequences of elements with indices from i to $j > i$ we borrow interval notation from computer aided design (CAD) system and write

$$a[i:j] = (a_i, \ldots, a_j),$$

but if finite bit strings are involved we write this as

Table 1: Logical connectives

$x \wedge y$	and
$x \vee y$	or
$/x$, $\overline{x}$	not
$x \oplus y$	exclusive or, + modulo 2

$$a[j:i] = (a_j, \ldots, a_i).$$

The set of all sequences of length n with elements from set A is denoted as A^n. The Hilbert epsilon operator picks an element εA from a set A. Applied to a singleton set it returns the unique element of the set:

$$\varepsilon\{x\} = x.$$

The cardinality of finite sets A is denoted by $\#A$. For symbols $x \in \mathbb{B}$ and natural numbers $n \in \mathbb{N}^+$, a bit-string obtained by repeating x exactly n times is defined as

$$x^n = \underbrace{x \ldots x}_{n \text{ times}} .$$

For bit strings $x \in \mathbb{B}^n$ we abbreviate the high order and low order (approximate) half of the bits as

$$\begin{aligned} x_H &= x[n-1 : \lfloor n/2 \rfloor] \\ x_L &= x[\lceil n/2 \rceil - 1 : 0]. \end{aligned}$$

1.2 Boolean Operators

In Boolean Algebra we use the connectives from Table 1. For logical connectives $\circ \in \{\wedge, \vee, \oplus\}$, bit-strings $a, b \in \mathbb{B}^n$, and a bit $c \in \mathbb{B}$, we borrow from vector calculus to define the corresponding bitwise operations:

$$\begin{aligned} /a(n-1:0) &= (/a_{n-1}, \ldots, /a_0) \\ a[n-1:0] \circ b[n-1:0] &= (a_{n-1} \circ b_{n-1}, \ldots, a_0 \circ b_0) \\ c \circ b[n-1:0] &= (c \circ b_{n-1}, \ldots, c \circ b_0). \end{aligned}$$

For records x with selectors $n_1, \ldots n_t$ we denote as usual with $x.n_i$ the component of the record selected by n_i. For subsequences $(n_{s_1}, \ldots, n_{s_r})$ of the selectors we abbreviate the sequence of record components selected by this subsequence as

$$x.(n_{s_1}, \ldots, n_{s_r}) = (x.n_{s_1}, \ldots, x.n_{s_r}).$$

In a hardware configuration h with components $h.pc, h.dpc, \ldots$ we would for instance abbreviate

$$h.(dpc, pc) = (h.dpc, h.pc).$$

1.3 Binary and Two's Complement Numbers

For bit-strings $a = a[n-1:0] \in \mathbb{B}^n$ we denote by

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

the interpretation of bit-string a as a *binary number*. String a is called the *binary representation* of length n of the natural number $\langle a \rangle$. It is often useful to decompose n bit binary representations $a[n-1:0]$ into an upper part $a[n-1:m]$ and a lower part $a[m-1:0]$. The correction between these parts is reflected in the following lemma (lemma 2.9 in [KMP14]).

Lemma 1 (decomposition). *Let $a \in \mathbb{B}^n$ and $n \geq m$. Then*

$$\langle a[n-1:0]\rangle = \langle a[n-1:m]\rangle \cdot 2^m + \langle a[m-1:0]\rangle.$$

The set of natural numbers representable as binary numbers of length n is denoted by

$$\mathbb{B}_n = \{\langle a\rangle \mid a \in \mathbb{B}^n\}.$$

For $x \in \mathbb{B}_n$ the binary representation of x of length n is denoted by

$$x_n = bin_n(x) = \varepsilon\{a \in \mathbb{B}^n \mid \langle a\rangle = x\}.$$

We denote by

$$[a] = -2^{n-1}a_{n-1} + \langle a[n-2:0]\rangle$$

the interpretation of bit-string a as a *two's complement number*. String a is called the *two's complement representation* of length n of the natural number $\langle a\rangle$. The set of natural numbers representable as two's complement numbers of length n is denoted by

$$\mathbb{T}_n = \{\langle a\rangle \mid a \in \mathbb{B}^n\}.$$

For $x \in \mathbb{T}_n$ the two's complement representation of x of length n is denoted by

$$twoc_n(x) = \varepsilon\{a \in \mathbb{B}^n \mid \langle a\rangle = x\}.$$

The binary addition $+_n$ and subtraction $-_n$ of bit strings $a, b \in \mathbb{B}^n$ is defined by

$$\begin{aligned} a +_n b &= bin_n((\langle a\rangle + \langle b\rangle) \bmod 2^n) \\ a -_n b &= bin_n((\langle a\rangle - \langle b\rangle) \bmod 2^n). \end{aligned}$$

A very easy computation shows, that for the addition of n bit numbers whose last m bits are all zero, it suffices to add the leading $n-m$ bits.

Lemma 2. *Let $a, b \in \mathbb{B}^n$ and let $a[m-1:0] = b[m-1:0] = 0^m$. Then*

$$a +_n b = (a[n-1:m] +_{n-m} b[n-1:m]) \circ 0^m.$$

As arithmetic units process binary numbers as well as two's complement numbers, one would expect in their specification also a counter part of this definition for two's complement numbers. However, in manuals such a specification is usually absent; instead addition and subtraction of two's complement numbers are also specified by the binary addition and subtraction operations $+_n$ and $-_n$. In a nutshell this works because according to lemma 2.14 of [KMP14] for $a \in \mathbb{B}^n$ we have

$$\langle a\rangle \equiv [a] \bmod 2^n$$

which immediately implies for $\circ \in \{+,-\}$:

$$\langle a\rangle \circ \langle b\rangle \equiv [a] \circ [b] \bmod 2^n.$$

This is almost but not quite what one wants to know. Attempting to define two's complement operators $\circ'_n$ one observes that the exact result

$$S = [a] \circ [b]$$

might lie outside of the range $\mathbb{T}_n$, just like $\langle a\rangle \circ \langle b\rangle$ might lie outside of $\mathbb{B}_n$. For such situations one replaces S by a number S tmod 2^n which is congruent to S modulo 2^n and which lies in the representable range.

$$S \text{ tmod } 2^n = \varepsilon\{x \in T_n \mid S \equiv x \bmod 2^n\}$$

Observe that the set on the right hand side of the equation is a singleton set, because congruence modulo 2^n is an equivalence relation with $\mathbb{T}_n$ as a system of representatives. The definition of two's complement addition and subtraction operators $\circ'_n$ then becomes

$$a \circ'_n b = twoc_n(([a] \circ [b]) \text{ tmod } 2^n).$$

The desired result (lemma 5.1 in [KMP14]) is then simply as follows.

Lemma 3.

$$a \circ_n b = a \circ'_n b$$

Proof oflemma 3. Let

$$s = a \circ_n b.$$

Then

$$\begin{aligned}[s] &\equiv \langle s \rangle \bmod 2^n \\ &\equiv \langle a \rangle \circ \langle b \rangle \bmod 2^n \\ &\equiv [a] \circ [b] \bmod 2^n.\end{aligned}$$

Trivially we have $[s] \in T_n$. Thus

$$[s] = ([a] \circ [b] \text{ tmod } 2^n). \qquad \square$$

1.4 Memory

The state of a byte addressable memory m with 32 address bits is modeled as a mapping

$$S : \mathbb{B}^{32} \to \mathbb{B}^8.$$

Such a memory will be used here in ISA specifications. For $x \in \mathbb{B}^{32}$ function value $m(x) \in \mathbb{B}^8$ models the current content of the memory at address x. The sequence $m_d(x)$ of d consecutive entries of memory S starting at address x is defined inductively as follows:

$$\begin{aligned}m_1(x) &= m(x) \\ m_{d+1}(x) &= m(x +_k d_k) \circ m_{d-1}(x).\end{aligned}$$

For bit strings $s \in \mathbb{B}^{8k}$, whose length is a multiple of 8, and $i < k$ we identify byte i of s as

$$byte(i,s) = s[8(i+1)-1 : 8i].$$

A trivial induction on d shows:

$$i < d \;\to\; byte(i, m_d(x)) = m(x +_{32} i_{32}). \tag{1}$$

The hardware memory systems we are going to construct will be addressable by cache lines which are 64 bits wide. The state resp. configuration of such a line addressable memory is thus modelled as a mapping

$$S : \mathbb{B}^{29} \to \mathbb{B}^{64}.$$

The set of all such configurations is denoted by K_m.

1.4.1 Embedding

We define a conversion function ℓ which changes byte addressable memories

$$m : \mathbb{B}^{32} \to \mathbb{B}^8$$

to their line addressable version.

$$\ell(m) : \mathbb{B}^{29} \to \mathbb{B}^{64}$$

For line addresses $a \in \mathbb{B}^{29}$ it is defined by

$$\ell(m)(a) = m_8(a \circ 000).$$

This is illustrated in Fig. 3. In processor correctness proofs it will serve as the simulation relation between the byte addressable ISA memory and the cache line addressable hardware memory system.

For the line addressable version of m we conclude the following.

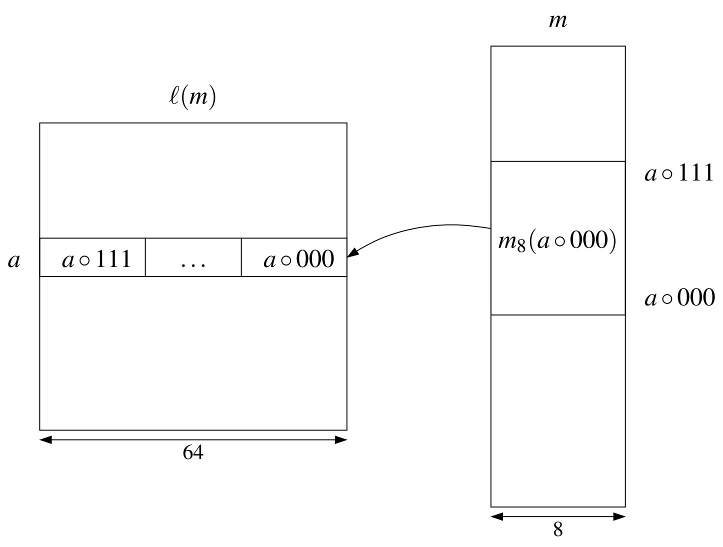


Fig. 3: Little endian embedding of byte-addressable memory into line-addressable memory

Lemma 4. *Assume $i < 8$.*

$$byte(i, \ell(m)(a)) = m(a \circ 000 +_{32} i_{32})$$

Proof of lemma 4.

$$\begin{aligned} byte(i, \ell(m)(a)) &= byte(i, m_8(a \circ 000)) && \text{(definition)} \\ &= m(a \circ 000 +_{32} i_{32}) && \text{(equation 1)} \end{aligned}$$ □

For byte addresses $x \in \mathbb{B}^{32}$, resp. for line addresses $x.l = x[31:3]$ and line offsets $x.o = x[2:0]$, we then have

$$m(x) = byte(\langle x.o \rangle, \ell(m)(x.l)).$$

1.4.2 Sequential Semantics

We define memory semantics for line addressable memory with the help of *accesses*. Such accesses $acc \in K_{acc}$ have the following components:

- processor address $acc.a \in \mathbb{B}^{29}$ (line address),
- processor data $acc.data \in \mathbb{B}^{64}$ — the input data in case of a write or a compare-and-swap (CAS),
- comparison data $acc.cdata \in \mathbb{B}^{32}$ — the data for comparison in case of a CAS access,
- the byte write signals $acc.bw \in \mathbb{B}^{8}$ for write and CAS accesses,
- write signal $acc.w \in \mathbb{B}$,
- read signal $acc.r \in \mathbb{B}$, and
- CAS signal $acc.cas \in \mathbb{B}$.

At most one of the bits w, r or cas is allowed to be on.

$$acc.r + acc.w + acc.cas \leq 1$$

In case none of these bits is on, we call the access *void*.

$$void(acc) \equiv acc.r + acc.w + acc.cas = 0$$

For technical reasons, we also require the byte write signals to be off in read accesses and to mask one of the words in case of CAS accesses:

$$\begin{aligned} acc.r &\rightarrow acc.bw = 0^8 \\ acc.cas &\rightarrow acc.bw \in \{0^4 1^4, 1^4 0^4\}. \end{aligned}$$

The set of all such accesses is denoted by K_{acc}. For CAS accesses and line addressable memory $m \in K_m$, we define the predicate $test(acc,m)$ which compares $acc.cdata$ with the upper or the lower word of the memory line $M(acc.a)$ addressed by the access, depending on the byte write signal $acc.bw[0]$.

$$test(acc,m) \equiv acc.cdata = \begin{cases} m(acc.a)_L & acc.bw[0] = 1 \\ m(acc.a)_H & acc.bw[0] = 0 \end{cases}$$

For $n = 64$ and strings $x, y \in \mathbb{B}^n$ function *modify* describes the replacement of bytes of x by the corresponding bytes of y under control of *byte write* signals $bw \in \mathbb{B}^8$.

$$byte(i, modify(x,y,bw)) = \begin{cases} byte(i,y) & bw[i] = 1 \\ byte(i,x) & bw[i] = 0 \end{cases}$$

Semantics of single accesses acc operating on a memory m is specified by a memory update function

$$\delta_M : K_m \times K_{acc} \rightarrow K_m$$

and the answers

$$dataout(m,acc) \in \mathbb{B}^{64}$$

of read and CAS accesses. Let

$$m' = \delta_M(m,acc).$$

Then memory is updated under control of signals bw, w and cas. The line addressed by $acc.a$ is updated in case of a write or in case of a CAS with positive outcome of the test. Only bytes $byte(i, m(acc.a))$ with active byte write signal $acc.bw_i$ are updated by the corresponding bytes of $acc.data$.

$$m'(a) = \begin{cases} modify(m(a), acc.data, acc.bw) & acc.a = a \wedge (acc.w \vee \\ & acc.cas \wedge test(acc,m)) \\ m(a) & \text{otherwise} \end{cases}$$

The answers $dataout(m,acc)$ of read or CAS accesses are defined as follows.

$$acc.r \vee acc.cas \rightarrow dataout(m,acc) = m(acc.a)$$

Obviously, void read accesses do not change the state of the memory and their answer is not specified. Overloading notation we consider linear access sequences

$$acc' : \mathbb{N} \rightarrow K_{acc}$$

where access $acc'(i)$ is access number i of the sequence. Execution of the first n accesses of such a sequence is then inductively defined by

$$\begin{aligned} \Delta_M^0(m,acc') &= m \\ \Delta_M^{i+1}(m,acc') &= \Delta_M(\Delta_M^i(m,acc'), acc'(i)). \end{aligned}$$

Readers familiar with [KMP14] will notice that accesses there had an extra component $acc.f \in \mathbb{B}$ whose activation signals so called *flush* accesses. These accesses necessarily have to be considered in the implementation of shared memory systems, but in the specification of such a system — and that is all we will rely on in this text — one can do without them.

2

Specification

The contents of this chapter are taken almost literally from the various sections of [LOP]. Thus, Sects. 2.1 and 2.2 — partially Sect. 4.1 of [LOP] — introduce basic MIPS ISA specification as well as semantics of the basic MIPS machine. Section 2.3 is exactly Sect. 7.1 of [LOP]; it extends the model of and integrates the interrupt mechanism into the basic machine model from Sect. 2.1. Finally, Sect. 2.4 is assembled from Sects. 9.1 and 9.2 of [LOP]; it presents the basic variant of virtual address translation. All these materials are included exclusively for completeness of presentation. The authorship over these materials belongs to all the authors of [LOP].

2.1 Basic MIPS

This is a fairly detailed summary and extension of results from Chap. 5 of [KMP14]. We go into considerable level of detail here, because i) the MIPS ISA is part of the formulation of every processor correctness theorem in this book and ii) *all* processor designs will be derived in one way or the other from the basic design. Understanding of future chapters will be greatly facilitated if readers have these specifications, constructions and arguments really at their fingertips.

The special purpose register file, move instructions and CAS instructions are already included in the basic design, because constructions and correctness proofs require no additional concepts. Changes in notation are marginal. Besides renaming of pins of units we only write the simulation relation between byte addressable ISA memory $c.m$ and line addressable hardware memory $h.m$ as

$$h.m = \ell(c.m)$$

instead of

$$h.m \sim c.m.$$

The introduction of instruction access $iacc(c)$ and data access $dacc(c)$ of ISA configurations have been moved forward from Chap. 9 of [KMP14], and the crucial result relating the update of byte addressable ISA memory $c.m$ with the update of its line addressable version $\ell(c.m)$ by data access $dacc(c)$ (lemma 9.3 of [KMP14]) is now formulated and proven in terms of ISA configurations c alone.

2.1.1 Instruction Tables

For the purpose of reference we include the full instruction tables from [Sch13b]. In contrast to [KMP14] we will already support move and CAS instructions in our basic processor designs. The treatment of the following instructions is deferred to later chapters: i) eret and sysc (interrupts), ii) mfence (store buffers), and iii) TLB instructions (address translation).

Table 2: R-type instructions

opcode	fun	rs	Mnemonic	Assembler-Syntax	Effect
Shift Operations					
000 000	000 000		sll	sll *rd rt sa*	rd = sll(rt,sa)
000 000	000 010		srl	srl *rd rt sa*	rd = srl(rt,sa)
000 000	000 011		sra	sra *rd rt sa*	rd = sra(rt,sa)
000 000	000 100		sllv	sllv *rd rt rs*	rd = sll(rt,rs)
000 000	000 110		srlv	srlv *rd rt rs*	rd = srl(rt,rs)
000 000	000 111		srav	srav *rd rt rs*	rd = sra(rt,rs)
Arithmetic, Logical Operations					
000 000	100 000		add	add *rd rs rt*	rd = rs $+$ rt
000 000	100 001		addu	addu *rd rs rt*	rd = rs $+$ rt
000 000	100 010		sub	sub *rd rs rt*	rd = rs $-$ rt
000 000	100 011		subu	subu *rd rs rt*	rd = rs $-$ rt
000 000	100 100		and	and *rd rs rt*	rd = rs $\wedge$ rt
000 000	100 101		or	or *rd rs rt*	rd = rs $\vee$ rt
000 000	100 110		xor	xor *rd rs rt*	rd = rs $\oplus$ rt
000 000	100 111		nor	nor *rd rs rt*	rd = $\overline{\text{rs} \vee \text{rt}}$
Test-and-Set Operations					
000 000	101 010		slt	slt *rd rs rt*	rd = (rs $<$ rt ? 1_{32} : 0_{32})
000 000	101 011		sltu	sltu *rd rs rt*	rd = (rs $<$ rt ? 1_{32} : 0_{32})
Jumps, System Call					
000 000	001 000		jr	jr *rs*	pc = rs
000 000	001 001		jalr	jalr *rd rs*	rd = pc + 4_{32}, pc = rs
000 000	001 100		sysc	sysc	System Call
Synchronizing Memory Operation					
000 000	111111		cas	cas *rd rt rd cdata*	rd'=m m'= (rd=cdata? rt: m)
Coprocessor Instructions					
opcode	fun		Mnemonic	Assembler-Syntax	Effect
010 000	011 000	10000	eret	eret	Exception Return
010 000		00100	movg2s	movg2s *rd rt*	spr[rd] := gpr[rt]
010 000		00000	movs2g	movs2g *rd rt*	gpr[rd] := spr[rt]
TLB Instructions					
000 000	111 101		flusht	flusht	flushes TLB translations
000 000	111 100		invlpg	invlpg *rs rt*	flushes TLB translations for addr. *rt* from ASID *rs*
Store Buffer Instruction					
000 000	111 110		mfence	mfence	flushes the SB

Table 3: J-type instructions

opcode	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j *iindex*	pc = bin_{32}(pc+4_{32})[31:28]iindex00

000 011	jal	jal *iindex*	R31 = pc + 4_{32},
			pc = bin_{32}(pc+4_{32})[31:28]iindex00

Table 4: I-type instructions. Note, we use the following shorthand ($d(c)$ is the access width).

$$\begin{aligned} \text{m} &= c.m_{d(c)}(ea(c)) \\ &= c.m_{d(c)}(c.gpr(rs(c)) +_{32} sxtimm(c)) \end{aligned}$$

opcode	rt	Mnemonic	Assembler-Syntax	Effect	Access Width
Data Transfer					
100 100		lbu	lbu *rt rs imm*	rt = 0^{24}m	1
100 101		lhu	lhu *rt rs imm*	rt = 0^{16}m	2
100 000		lb	lb *rt rs imm*	rt = sxt(m)	1
100 001		lh	lh *rt rs imm*	rt = sxt(m)	2
100 011		lw	lw *rt rs imm*	rt = m	4
101 000		sb	sb *rt rs imm*	m = rt[7:0]	1
101 001		sh	sh *rt rs imm*	m = rt[15:0]	2
101 011		sw	sw *rt rs imm*	m = rt	4
Arithmetic, Logical Operation, Test-and-Set					
001 000		addi	addi *rt rs imm*	rt = rs + sxt(imm)	
001 001		addiu	addiu *rt rs imm*	rt = rs + sxt(imm)	
001 010		slti	slti *rt rs imm*	rt = (rs $<$ sxt(imm) ? 1_{32} : 0_{32})	
001 011		sltiu	sltiu *rt rs imm*	rt = (rs $<$ sxt(imm) ? 1_{32} : 0_{32})	
001 100		andi	andi *rt rs imm*	rt = rs $\wedge$ zxt(imm)	
001 101		ori	ori *rt rs imm*	rt = rs $\vee$ zxt(imm)	
001 110		xori	xori *rt rs imm*	rt = rs $\oplus$ zxt(imm)	
001 111		lui	lui *rt imm*	rt = imm0^{16}	
Branch					
000 001	00000	bltz	bltz *rs imm*	pc = pc $+$ (rs $<$ 0 ? imm00 : 4_{32})	
000 001	00001	bgez	bgez *rs imm*	pc = pc $+$ (rs $\geq$ 0 ? imm00 : 4_{32})	
000 100		beq	beq *rs rt imm*	pc = pc $+$ (rs $=$ rt ? imm00 : 4_{32})	
000 101		bne	bne *rs rt imm*	pc = pc $+$ (rs $\neq$ rt ? imm00 : 4_{32})	
000 110	00000	blez	blez *rs imm*	pc = pc $+$ (rs $\leq$ 0 ? imm00 : 4_{32})	
000 111	00000	bgtz	bgtz *rs imm*	pc = pc $+$ (rs $>$ 0 ? imm00 : 4_{32})	

2.1.2 Configuration and Instruction Fields

A basic *MIPS configuration c* has four user visible data structures (Fig. 4):

- $c.pc \in \mathbb{B}^{32}$ — the program counter (PC).
- $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$ — the general purpose register (GPR) file consisting of 32 registers, each 32 bits wide. Register number zero is tied to 0_{32}. Writing to it will have no effect.

$$c.gpr(0_5) = 0_{32}$$

- $c.m : \mathbb{B}^{32} \to \mathbb{B}^8$ — the processor memory. It is byte addressable; addresses have 32 bits.
- $c.spr : \mathbb{B}^5 \to \mathbb{B}^{32}$ — the special purpose register (SPR) file.

Program counter and general purpose registers belong to the central processing unit (CPU). Let K be the set of all basic MIPS configurations. A mathematical definition of the ISA will be given by a function

$$\delta_{isa} : K \to K$$

where

$$c' = \delta_{isa}(c)$$

is the configuration reached from configuration c, if the next instruction is executed. An ISA computation is a sequence (c^i) of ISA configurations with $i \in \mathbb{N}$ satisfying

$$\begin{aligned} c^0.pc &= 0^{32} \\ c^{i+1} &= \delta_{isa}(c^i), \end{aligned}$$

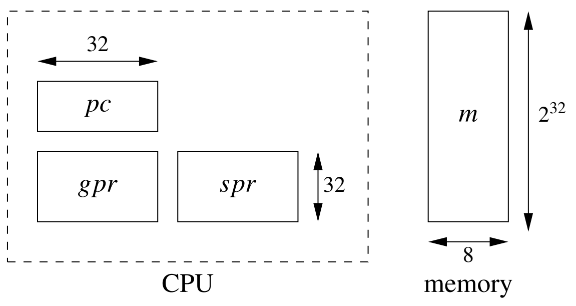


Fig. 4: Visible data structures of MIPS ISA

i.e., initially the program counter points to address 0^{32} and in each step one instruction is executed. In the remainder of this section we specify the ISA simply by specifying function δ_{isa}, i.e., by specifying

$$c' = \delta_{isa}(c)$$

for all configurations c. Recall, in Sect. 1.3 for numbers $y \in \mathbb{B}^n$ we abbreviated the binary representation of y with n bits as

$$y_n = bin_n(y)$$

and for memories

$$m : \mathbb{B}^{32} \to \mathbb{B}^8,$$

addresses $a \in \mathbb{B}^{32}$, and numbers d of bytes, we denote the content of d consecutive memory bytes starting at address a by $m_d(a)$.

The current instruction $I(c)$ to be executed in configuration c is defined by the 4 bytes in memory addressed by the current program counter:

$$I(c) = c.m_4(c.pc).$$

Because instructions are 4 bytes long, we require for the time being that instructions are *aligned* at and fetched from 4 byte boundaries:

$$c.pc[1:0] = 00.$$

When we treat interrupts, we will raise a misalignment interrupt if this condition is violated.

The six high order bits of the current instruction are called the opcode:

$$opc(c) = I(c)[31:26].$$

There are three instruction types: R-, J-, and I-type. The current instruction type is determined by the following predicates:

$$\begin{aligned} rtype(c) &\equiv opc(c) = 0{*}0^4 \\ jtype(c) &\equiv opc(c) = 0^4 1{*} \\ itype(c) &= /rtype(c) \wedge /jtype(c). \end{aligned}$$

Depending on the instruction type, the bits of the current instruction are subdivided as shown in Fig. 5. Register addresses are specified in the following fields of the current instruction:

$$\begin{aligned} rs(c) &= I(c)[25:21] \\ rt(c) &= I(c)[20:16] \\ rd(c) &= I(c)[15:11]. \end{aligned}$$

For R-type instructions, ALU-functions to be applied to the register operands can be specified in the function field:

$$fun(c) = I(c)[5:0].$$

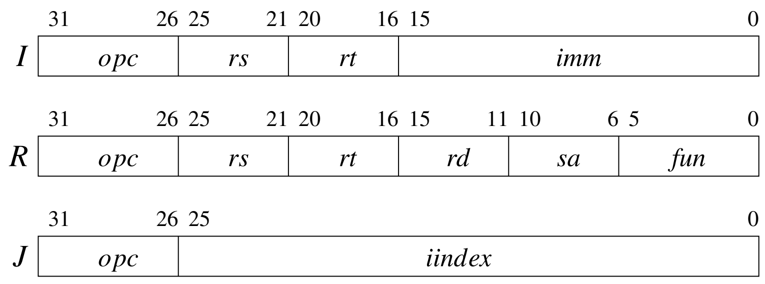


Fig. 5: Types and fields of MIPS instructions

Three kinds of immediate constants are specified: the shift amount *sa* in R-type instructions, the immediate constant *imm* in I-type instructions, and an instruction index *iindex* in J-type (like jump) operations:

$$\begin{aligned} sa(c) &= I(c)[10:6] \\ imm(c) &= I(c)[15:0] \\ iindex(c) &= I(c)[25:0]. \end{aligned}$$

Immediate constant *imm* has 16 bits. In order to apply ALU functions to it, the constant can be extended with 16 high order bits in two ways: zero extension and sign extension:

$$\begin{aligned} zxtimm(c) &= 0^{16}imm(c) \\ sxtimm(c) &= imm(c)[15]^{16}imm(c) \\ &= I(c)[15]^{16}imm(c). \end{aligned}$$

2.1.3 Instruction Decoding

For every mnemonic *mn* of a MIPS instruction from the tables above, we define a predicate $mn(c)$ which is true, if instruction *mn* is to be executed in configuration c. For instance,

$$\begin{aligned} lw(c) &\equiv opc(c) = 100011 \\ bltz(c) &\equiv opc(c) = 0^5 1 \wedge rt(c) = 0^5 \\ add(c) &\equiv rtype(c) \wedge fun(c) = 10^5. \end{aligned}$$

The remaining predicates directly associated to the mnemonics of the assembly language are derived in the same way from the tables. We group the basic instruction set into six groups and define for each group a predicate that holds, if an instruction from that group is to be executed.

- ALU-operations of I-type are recognized by the leading three bits of the opcode, resp. $I(c)[31:29]$; ALU-operations of R-type — by the two leading bits of the function code, resp. $I(c)[5:4]$:

$$\begin{aligned} alur(c) &\equiv rtype(c) \wedge fun(c)[5:4] = 10 \\ alui(c) &\equiv itype(c) \wedge opc(c)[5:3] = 001 \\ alu(c) &= alur(c) \vee alui(c). \end{aligned}$$

- Shift unit operations are of R-type and are recognized by the three leading bits of the function code. If bit $fun(c)[2]$ of the function code is on, the shift distance is taken from register specified by $rs(c)$:[1]

$$\begin{aligned} su(c) &\equiv rtype(c) \wedge fun(c)[5:3] = 000 \\ suv(c) &\equiv su(c) \wedge fun(c)[2]. \end{aligned}$$

[1] Mnemonics with suffix *v* as "variable"; one would expect instead for the other shifts a suffix *i* as "immediate".

- Loads and stores are of I-type and are recognized by the three leading bits of the opcode. CAS are of R-type and are recognized by all ones in the opcode.

$$\begin{aligned} l(c) &\equiv itype(c) \wedge opc(c)[5:3] = 100 \\ s(c) &\equiv itype(c) \wedge opc(c)[5:3] = 101 \\ cas(c) &\equiv rtype(c) \wedge opc(c) = 1^6 \end{aligned}$$

  Memory operations are loads, stores, or CAS. For convenience we introduce the following shorthands.

$$\begin{aligned} ls(c) &= l(c) \vee s(c) \\ mop(c) &= ls(c) \vee cas(c) \end{aligned}$$

- Branches are of I-type and are recognized by the three leading bits of the opcode:

$$\begin{aligned} b(c) &\equiv itype(c) \wedge opc(c)[5:3] = 000 \\ &\equiv itype(c) \wedge I(c)[31:29] = 000. \end{aligned}$$

- Jumps are defined in a brute force way:

$$\begin{aligned} jump(c) &= jr(c) \vee jalr(c) \vee j(c) \vee jal(c) \\ jb(c) &= jump(c) \vee b(c). \end{aligned}$$

- Moves are from GPR to SPR or vice versa.

$$move(c) = movg2s(c) \vee movs2g(c)$$

2.1.4 Processor-Local Operations

Reading out Data from Register Files

Addressing the general purpose register file $c.gpr$ with $rs(c)$ and $rt(c)$ and the special register file $c.spr$ with $rs(c)$ we obtain shorthands for register file contents:

$$\begin{aligned} A(c) &= c.gpr(rs(c)) \\ B(c) &= c.gpr(rt(c)) \\ S(c) &= c.spr(rs(c)). \end{aligned}$$

Moves

Instruction $movg2s$ writes $B(c)$ into the special purpose register file at address $rd(s)$. Instruction $movs2g$ writes $S(c)$ into the special purpose register file at address $rd(c)$. The memory and other register file contents are not changed. The pc is incremented by 4. A summary of move operations ($mov(c)$) is then

$$\begin{aligned} c'.pc &= c.pc +_{32} 4_{32} \\ c'.gpr(x) &= \begin{cases} S(c) & movs2g(c) \wedge x = rd(c) \wedge x \neq 0_5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr(x) &= \begin{cases} B(c) & movg2s(c) \wedge x = rd(c) \\ c.spr(x) & \text{otherwise} \end{cases} \\ c'.m &= c.m. \end{aligned}$$

Table 5: Specification of ALU operations. Result $alu.res \in \mathbb{B}^n$ and overflow bit $alu.ovf \in \mathbb{B}$ are given for ALU operands $alu.a, alu.b \in \mathbb{B}^n$, function bits $alu.f \in \mathbb{B}^4$, and special bit $alu.i \in \mathbb{B}$.

$f[3:0]$	i	res	ovf
0000	$*$	$a +_n b$	$[a]+[b] \notin T_n$
0001	$*$	$a +_n b$	0
0010	$*$	$a -_n b$	$[a]-[b] \notin T_n$
0011	$*$	$a -_n b$	0
0100	$*$	$a \wedge b$	0
0101	$*$	$a \vee b$	0
0110	$*$	$a \oplus b$	0
0111	0	$\overline{a \vee b}$	0
0111	1	$b[n/2-1:0] \circ 0^{n/2}$	0
1010	$*$	$0^{n-1} \circ ([a] < [b]\ ?\ 1 : 0)$	0
1011	$*$	$0^{n-1} \circ (\langle a \rangle < \langle b \rangle\ ?\ 1 : 0)$	0

ALU-operations

ALU operations are defined with the help of Table 5. It defines functions $res(a,b,f,i)$ and $ovf(a,b,f,i)$. As we do not treat interrupts yet, we use only the first of these functions here. We observe that in all ALU operation of the ISA the left operand is always

$$alu.a(c) = A(c).$$

For R-type instruction the right operand is the register specified by the *rt* field. For I-type instructions it is the sign extended immediate operand if

$$opc(c)[2] = I(c)[28] = 0$$

or zero extended immediate operand if

$$opc(c)[2] = 1.$$

Thus, we define immediate fill bit $ifill(c)$, extended immediate constant $xtimm(c)$, and right operand $alu.b(c)$ in the following way:

$$\begin{aligned} ifill(c) &= \begin{cases} imm(c)[15] & opc(c)[2] = 0 \\ 0 & opc(c)[2] = 1 \end{cases} \\ xtimm(c) &= \begin{cases} sxtimm(c) & opc(c)[2] = 0 \\ zxtimm(c) & opc(c)[2] = 1 \end{cases} \\ &= ifill(c)^{16} imm(c) \\ alu.b(c) &= \begin{cases} B(c) & rtype(c) \\ xtimm(c) & \text{otherwise.} \end{cases} \end{aligned}$$

Comparing Table 5 with the tables for I-type and R-type instructions we see that bits $af[2:0]$ of the ALU control can be taken from the low order fields of the opcode for I-type instructions and from the low order bits of the function field for R-type instructions:

$$alu.f(c)[2:0] = \begin{cases} fun(c)[2:0] & rtype(c) \\ opc(c)[2:0] & \text{otherwise.} \end{cases}$$

For bit $alu.f[3]$ things are more complicated. For R-type instructions it can be taken from the function code. For I-type instructions it must only be forced to 1 for the two test and set operations, which can be recognized by

Table 6: Specification of shift unit operations. Result $su.res \in \mathbb{B}^n$ of the logical left shift sll, logical right shift srl, and arithmetic right shift sra, is computed for shift operand $su.b \in \mathbb{B}^n$ and (binary coded) shift distance $\langle su.dist \rangle \in [0 : n-1]$ under control of the function bits $su.f \in \mathbb{B}^2$. Results of various shifts of operand $su.b$ by distance i are given in the rightmost column.

$f[1:0]$	res	res for $\langle dist \rangle = i$
00	$sll(b, \langle dist \rangle)$	$b[n-i-1:0] \circ 0^i$
10	$srl(b, \langle dist \rangle)$	$0^i \circ b[n-1:i]$
11	$sra(b, \langle dist \rangle)$	$b_{n-1}^i \circ b[n-1:i]$

$$opc(c)[2:1] = 01.$$

$$alu.f(c)[3] = \begin{cases} fun(c)[3] & rtype(c) \\ \overline{opc(c)[2]} \wedge opc(c)[1] & \text{otherwise} \end{cases}$$

The i-input of the ALU distinguishes for

$$af[3:0] = 0111$$

between the lui-instruction of I-type for $i = 0$ and the nor-instruction of R-type for $i = 1$. Thus, we set it to $itype(c)$. The result of the ALU and the arithmetic overflow computed with these inputs are denoted resp. by

$$alu.res(c) = alu.res(alu.a(c), alu.b(c), alu.f(c), itype(c))$$

and

$$alu.ovf(c) = alu.ovf(alu.a(c), alu.b(c), alu.f(c), itype(c)).$$

Depending on the instruction type, the destination register $rdes$ is specified by the rd field or the rt field:

$$rdes(c) = \begin{cases} rd(c) & rtype(c) \\ rt(c) & \text{otherwise.} \end{cases}$$

A summary of *all* ALU operations ($alu(c)$) is then

$$\begin{aligned} c'.pc &= c.pc +_{32} 4_{32} \\ c'.gpr(x) &= \begin{cases} alu.res(c) & x = rdes(c) \wedge x \neq 0_5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr &= c.spr \\ c'.m &= c.m. \end{aligned}$$

Shift Unit Operations

Results of shift unit operations is defined with the help of Table 6 as a function $res(b, dist, f)$. They come in two flavors. If $fun(c)[2]$ is set, the shift distance $su.dist(c)$ is specified by the last bits of the register specified by the rs field. Otherwise the shift distance is an immediate operand specified by the sa field of the instruction.

$$su.dist(c) = \begin{cases} A(c)[4:0] & fun(c)[2] = 1 \\ sa(c) & fun(c)[2] = 0 \end{cases}$$

The left operand that is shifted is always the register specified by the rt field:

$$su.b(c) = B(c)$$

and the control bits $su.f[1:0]$ are taken from the low order bits of the function field:

$$su.f(c) = fun(c)[1:0].$$

The result of the shift unit computed with these inputs is denoted by

$$su.res(c) = su.res(su.bc, su.dist(c), su.f(c)).$$

For shift operations the destination register is always specified by the *rd* field. Thus, the shift unit operations ($su(c)$) can be summarized as

$$\begin{aligned}
c'.pc &= c.pc +_{32} 4_{32}\\
c'.gpr(x) &= \begin{cases} su.res(c) & x = rd(c) \wedge x \neq 0_5\\ c.gpr(x) & \text{otherwise}\end{cases}\\
c'.spr &= c.spr\\
c'.m &= c.m.
\end{aligned}$$

Branch and Jump

A branch condition evaluation unit was specified in Table 7. It computes a function $res(a,b,f)$. We use this function with the following parameters:

$$\begin{aligned}
bce.a(c) &= A(c)\\
bce.b(c) &= B(c)\\
bce.f(c) &= opc(c)[2:0] \circ rt(c)[0]
\end{aligned}$$

and define the result of a branch condition evaluation as

$$bce.res(c) = bce.res(bce.a(c), bce.b(c), bce.f(c)).$$

The next program counter $c'.pc$ is usually computed as $c.pc +_{32} 4_{32}$. This order is only changed in jump instructions or in branch instructions, where the branch is taken, i.e., the branch condition evaluates to 1. We define

$$jbtaken(c) = jump(c) \vee b(c) \wedge bce.res(c).$$

In case of a jump or a branch taken, there are three possible jump targets (see below).

i) Branch instructions involve a *relative* branch.

$$b(c) \wedge bce.res(c)$$

The *pc* is incremented by a branch distance:

$$\begin{aligned}
bdist(c) &= imm(c)[15]^{14} \circ imm(c) \circ 00\\
btarget(c) &= c.pc +_{32} bdist(c).
\end{aligned}$$

Note that the branch distance is a kind of a sign extended immediate constant, but due to the alignment requirement the two low order bits of the jump distance must be zeros. Thus, one uses the 16 bits of the immediate constant for bits $[17:2]$ of the jump distance. Sign extension is used for the remaining bits. Thus, backward jumps are realized with negative $[imm(c)]$.

ii) R-type jumps

$$jr(c) \vee jalr(c).$$

The branch target is specified by the *rs* field of the instruction:

$$btarget(c) = A(c).$$

Table 7: Specification of branch condition evaluation. Result $bce.res \in \mathbb{B}$ is given for operands $bce.a, bce.b \in \mathbb{B}^n$ and function bits $bce.f \in \mathbb{B}^4$.

$f[3:0]$	res
0010	$[a] < 0$
0011	$[a] \geq 0$
100*	$a = b$
101*	$a \neq b$
110*	$[a] \leq 0$
111*	$[a] > 0$

iii) J-type jumps

$$j(c) \vee jal(c).$$

The branch target is computed in a rather peculiar way: i) the *pc* is incremented by 4, ii) *then* bits $[27:2]$ are replaced by the *iindex* field of the instruction:

$$btarget(c) = (c.pc +_{32} 4_{32})[31:28] \circ iindex(c) \circ 00.$$

Now we can define the next *pc* computation for *all* instructions as

$$\begin{aligned} btarget(c) &= \begin{cases} c.pc +_{32} bdist(c) & b(c) \wedge bce.res(c) \\ A(c) & jr(c) \vee jalr(c) \\ (c.pc +_{32} 4_{32})[31:28] \circ iindex(c) \circ 00 & \text{otherwise} \end{cases} \\ c'.pc &= nextpc(c) \\ &= \begin{cases} btarget(c) & jbtaken(c) \\ c.pc +_{32} 4_{32} & \text{otherwise.} \end{cases} \end{aligned}$$

Jump and Link

Jump and link instructions

$$jal(c) \vee jalr(c)$$

are used to implement calls of procedures. Besides setting the *pc* to the branch target, they prepare the so called *link address*

$$linkad(c) = c.pc +_{32} 4_{32}$$

and save it in a register. For the R-type instruction *jalr*, this register is specified by the *rd* field. J-type instruction *jal* does not have an *rs* field, and the link address is stored in register number 31 ($\langle 1^5 \rangle$). Branch and jump instructions do not change the memory. Therefore, for the update of registers in branch and jump instructions

$$jb(c)$$

we have:

$$\begin{aligned} c'.gpr(x) &= \begin{cases} linkad(c) & jalr(c) \wedge x = rd(c) \wedge x \neq 0_5 \vee jal(c) \wedge x = 1^5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr &= c.spr \\ c'.m &= c.m. \end{aligned}$$

2.1.5 Memory Operations

Memory operations access a certain number

$$d(c) \in \{1,2,4\}$$

of bytes of memory starting at a so called *effective address* $ea(c)$. Letters b, h, and w in the mnemonics define the width:

- b stands for $d = 1$ resp. a byte access;
- h stands for $d = 2$ resp. a half word access, and
- w stands for $d = 4$ resp. a word access.

Inspection of the instruction tables gives

$$d(c) = \begin{cases} 1 & opc(c)[0] = 0 \\ 2 & opc(c)[1:0] = 01 \\ 4 & opc(c)[1:0] = 11 \vee cas(c). \end{cases}$$

Addressing is always relative to $A(c)$. Except for CAS operations (which have R-type) the offset is specified by the immediate field:

$$ea(c) = \begin{cases} A(c) & cas(c) \\ A(c) +_{32} sxtimm(c) & \text{otherwise} \end{cases}$$

Note that the immediate constant is sign extended. Thus, negative offsets can be realized in the same way as negative branch distances. In the absence of misalignment interrupts addresses are for the time being required to be *aligned.* If we interpret them as binary numbers they have to be divisible by the width $d(c)$:

$$d(c) \mid \langle ea(c) \rangle$$

or equivalently

$$\begin{aligned} mop(c) \wedge d(c) = 4 \rightarrow \ & ea(c)[1:0] = 00 \\ mop(c) \wedge d(c) = 2 \rightarrow \ & ea(c)[0] = 0. \end{aligned}$$

Stores

A store instruction takes the low order $d(c)$ bytes of $B(c)$ and stores them as $m_{d(c)}(ea(c))$. The pc is incremented by 4 (but we have already defined that on page 24). Other memory bytes and register values are not changed.

$$\begin{aligned} c'.gpr &= c.gpr \\ c'.spr &= c.spr \\ c'.m(x) &= \begin{cases} byte(i, B(c)) & x = ea(c) +_{32} i_{32} \wedge i < d(c) \\ c.m(x) & \text{otherwise} \end{cases} \end{aligned}$$

Loads

Loads, like stores, access $d(c)$ bytes of memory starting at address $ea(c)$. The result is stored in the low order $d(c)$ bytes of the destination register, which is specified by the rt field of the instruction. This leaves

$$32 - 8 \cdot d(c)$$

bits of the destination register to be filled by some bit *fill*(c). For unsigned loads (with a suffix "u" in the mnemonics) the fill bit is zero; otherwise it is sign extended by the leading bit of

$$c.m_{d(c)}(ea(c)).$$

In this way a load result

$$lres(c) \in \mathbb{B}^{32}$$

is computed

$$\begin{aligned} u(c) &= opc(c)[2] \\ fill(c) &= \begin{cases} 0 & u(c) \\ c.m(ea(c) +_{32} (d(c)-1)_{32})[7] & \text{otherwise} \end{cases} \\ lres(c) &= fill(c)^{32-8\cdot d(c)} c.m_{d(c)}(ea(c)) \end{aligned}$$

and the general purpose register specified by the *rt* field is updated. Other registers and the memory are left unchanged.

$$\begin{aligned} c'.gpr(x) &= \begin{cases} lres(c) & x = rt(c) \wedge x \neq 0_5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr &= c.spr \\ c'.m &= c.m \end{aligned}$$

Compare and Swap (CAS)

Recall that CAS operations have R-type, thus no immediate constant is available and the effective address is just

$$ea(c) = A(c).$$

A load word operation with destination specified by the *rd* field is performed. The content of the ninth register of the special purpose register file

$$cdata(c) = c.spr(9_5)$$

is compared with the memory word at the effective address

$$castest(c) \equiv cdata(c) = c.m_4(ea(c)).$$

If the test is positive, the memory content is replaced by $B(c)$.

$$\begin{aligned} c'.gpr(x) &= \begin{cases} c.m_4(ea(c)) & x = rd(c) \wedge x \neq 0_5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr &= c.spr \\ c'.m(x) &= \begin{cases} byte(i, B(c)) & castest(c) \wedge x = ea(c) +_{32} i_{32} \wedge i < 4 \\ c.m(x) & \text{otherwise} \end{cases} \end{aligned}$$

For convenience, we introduce shorthands to classify the memory operations. Loads and CAS read from memory. Stores and "positive" CAS write to memory. Thus CAS is both reading and (potentially) writing.

$$\begin{aligned} read(c) &= l(c) \vee cas(c) \\ write(c) &= s(c) \vee cas(c) \wedge castest(c) \end{aligned}$$

2.2 Summary

For convenience, in this section we gather in one place all key parts defining the specification machine and its computation (ISA computation):

i) MIPS ISA (in Sect. 2.2.1), presenting the material of Sect. 2.1 basically on one page,
ii) software conditions (in Sect. 2.2.2) under which the ISA above is meaningful, and
iii) accesses of the ISA computation (in Sect. 2.2.3), constructed according to the definitions from Sect. 1.4.2.

2.2.1 MIPS ISA

We collect all previous definitions of destination registers for the general purpose register file and the special purpose register file into

$$xad(c) = \begin{cases} 1^5 & jal(c) \\ rd(c) & rtype(c) \\ rt(c) & \text{otherwise.} \end{cases}$$

Also we collect the data *gpr.in* to be written into the general purpose register file.

$$gpr.in(c) = \begin{cases} lres(c) & read(c) \\ C(c) & \text{otherwise} \end{cases}$$

For technical reasons, we define on the way an intermediate result C that collects the possible GPR input from arithmetic, shift, and jump instructions:

$$C(c) = \begin{cases} S(c) & movs2g(c) \\ B(c) & movg2s(c) \\ su.res(c) & su(c) \\ linkad(c) & jal(c) \vee jalr(c) \\ alu.res(c) & \text{otherwise.} \end{cases}$$

Finally, we collect in a general purpose register write signal all occasions when some general purpose register is updated:

$$gpr.w(c) = alu(c) \vee su(c) \vee read(c) \vee jal(c) \vee jalr(c).$$

Now we can summarize the MIPS ISA in three rules concerning the updates of *pc*, general purpose registers, and memory.

$$\begin{aligned} c'.pc &= \begin{cases} btarget(c) & jbtaken(c) \\ c.pc +_{32} 4_{32} & \text{otherwise} \end{cases} \\ c'.gpr(x) &= \begin{cases} gpr.in(c) & gpr.w(c) \wedge x = xad(c) \wedge x \neq 0_5 \\ c.gpr(x) & \text{otherwise} \end{cases} \\ c'.spr(x) &= \begin{cases} C(c) & movg2s(c) \wedge x = xad(c) \\ c.spr(x) & \text{otherwise} \end{cases} \\ c'.m(x) &= \begin{cases} byte(i, B(c)) & write(c) \wedge x = ea(c) +_{32} i_{32} \wedge i < d(c) \\ c.m(x) & \text{otherwise} \end{cases} \end{aligned}$$

2.2.2 Software Conditions

In the absence of misalignment interrupts we have already required that instruction fetch and memory operations are aligned.

$$\begin{aligned} & 4 \mid \langle c.pc \rangle \\ mop(c) \; \rightarrow \; & d(c) \mid \langle ea(c) \rangle \end{aligned}$$

Most of ISA memory will be implemented by some kind of RAM, which happens to have unknown content after power up. The program counter points initially to address 0_{32} and starts fetching instructions from some initial program, for instance a boot loader. This program will reside in ROM occupying the low order 2^{r+3} byte addresses of the memory system for some r. Because write operations to ROM have no effect, we simply forbid store or CAS operations to that region.

$$write(c) \;\rightarrow\; \langle ea(c)\rangle \geq 2^{r+3}$$

Note, in the presence of guard conditions (see Sect. 2.4.3) the software conditions above are guaranteed to hold only in case the corresponding computation is *guarded*, i.e., respects all *related* guard conditions. As formally described in [Obe17], execution of instructions is inherently split into two phases: *fetch* and *execute*. Within the fetch phase, all ISA signals necessary to fetch the current instruction ($I(c)$) are computed, and the phase ends with a fetch of the instruction word. The remaining computations performed to complete execution of the current instruction constitute the executed phase, which of course ends with the current instruction finishing execution.

All guard conditions imposed on steps of the processor core executing instructions are of course split into two categories: those that restrict computations performed in the fetch phase, and all the remaining ones, that restrict computations performed in the execute phase. Therefore, we require the software conditions which apply in the fetch phase of computational step n to hold *if* the guard conditions are obeyed by all steps before n as well as by all computations performed in the fetch phase of step n. Clearly, all the remaining software conditions which apply to step n must hold only *if* the guard conditions are obeyed by all steps up to n (before and within step n).

The assumptions above turn out to be crucial when one tries to justify correctness of pipelined machines. Thus, in Chap. 9 we actually rely on the software conditions for the fetch phase in order to verify the guard conditions for the execute phase (see Sect. 9.4.2).

2.2.3 Accesses of ISA

As a guideline for the construction of environments $sh4s$ and $sh4l$ with their shifters supporting the memory operations of the ISA, we rephrase these operations in terms of data accesses $dacc(c)$ to the line addressable version $\ell(c.m)$ of ISA memory. In a completely straightforward way one specifies

$$\begin{aligned} dacc(c).a &= ea(c).l \\ dacc(c).cdata &= cdata(c) \\ dacc(c).r &= l(c) \\ dacc(c).w &= s(c) \\ dacc(c).cas &= cas(c) \end{aligned}$$

For the construction of the data memory input $dmin(c)$ of store or CAS operations one shifts the register $B(c)$, whose low order $d(c) \leq 4$ bytes are to be stored, by $\langle ea(c)[1:0]\rangle$ bytes to the left and then makes two copies of the result F; which copy is used will be determined by the byte write signals.

$$\begin{aligned} F(c) &= slc(B(c), \langle ea(c)[1:0]\rangle) \\ dmin(c) &= F(c) \circ F(c) \end{aligned}$$

For the byte write signals bw — and also for byte read signals br to be used later for store buffer forwarding — one constructs first a 4 bit wide mask $mmask$ for memory operations with $d(c)$ many ones at the right end. This mask is shifted left by $\langle ea(c)\rangle$. Two copies of the result $f(c)$ are produced. The left copy is used in the byte read and byte write signals if $ea(c)[2]$ is on; otherwise the right copy is used.

$$\begin{aligned} mmask(c) &= mop(c) \wedge (0^{4-d(c)}1^{d(c)}) \\ f(c) &= slc(mmask(c), \langle ea(c)[1:0]\rangle) \\ br(c) &= l(c) \wedge (ea[2] \wedge f(c)) \circ (\overline{ea[2]} \wedge f(c)) \\ bw(c) &= (s(c) \vee cas(c)) \wedge (ea[2] \wedge f(c)) \circ (\overline{ea[2]} \wedge f(c)) \end{aligned}$$

Note that byte read signals are not generated for CAS instructions because these instructions will not use store buffer forwarding. Using the definition above we specify:

$$
\begin{aligned}
dacc(c).data &= dmin(c)\\
dacc(c).bw &= bw(c).
\end{aligned}
$$

As the output of the data access we have

$$dmout(c) = dataout(\ell(c.m), dacc(c)).$$

For $i < d(c)$ we locate the bytes of B to be stored in the data memory input by

$$
\begin{aligned}
byte(i, B(c)) &= byte(i + \langle ea(c)[1:0]\rangle, F(c))\\
&= byte(i + \langle 0 \circ ea(c)[1:0]\rangle, dmin(c))\\
&= byte(i + \langle 1 \circ ea(c)[1:0]\rangle, dmin(c))\\
&= byte(i + \langle ea(c)[2:0]\rangle, dmin(c)).
\end{aligned}
$$

Similarly, for $i < d(c)$ we locate the byte write and byte read signals of the bytes to be stored in $bw[7:0]$ resp. $br[7:0]$ by

$$
\begin{aligned}
f(c)[j] = 1 &\leftrightarrow mop(c) \wedge \exists i < d(c) : j = \langle ea(c)[1:0]\rangle + i\\
bw(c)[j] = 1 &\leftrightarrow (s(c) \vee cas(c)) \wedge \exists i < d(c) : j = \langle ea(c)[2:0]\rangle + i\\
br(c)[j] = 1 &\leftrightarrow l(c) \wedge \exists i < d(c) : j = \langle ea(c)[2:0]\rangle + i\,.
\end{aligned}
$$

For later reference we summarize these arguments in the following lemma.

Lemma 5.

$$
\begin{aligned}
byte(i, B(c)) &= byte(i + \langle ea(c)[2:0]\rangle, dmin(c))\\
bw(c)[j] = 1 &\leftrightarrow (s(c) \vee cas(c)) \wedge \exists i < d(c) : j = \langle ea(c)[2:0]\rangle + i
\end{aligned}
$$

We prove the crucial result, that the line addressable version $\ell(c'.m)$ of the next ISA configuration is obtained by applying the data access $dacc(c)$ defined in this way to $\ell(c.m)$ (which is what the hardware memory $h.m$ is specified to do).

Lemma 6.

$$\ell(c'.m) = \delta_M(\ell(c.m), dacc(c))$$

Proof of lemma 6. Abbreviating the right hand side

$$M' = \delta_M(\ell(c.m), dacc(c))$$

and applying the definition of memory semantics δ_M we get

$$
M'(a) = \begin{cases}
modify(\ell(c.m)(a), dmin(c), bw(c)) & a = dacc(c).a \wedge (dacc(c).w \vee \\
 & dacc(c).cas \wedge test(dacc(c), \ell(c.m))\\
\ell(c.m(a)) & \text{otherwise.}
\end{cases}
$$

For CAS accesses ($dacc(c).cas = 1$) we have:

$$
\begin{aligned}
test(dacc(c), \ell(c.m)) &\equiv dacc(c).cdata = \begin{cases} \ell(c.m)(dacc(c).a)_H & dacc(c).bw[0]\\ \ell(c.m)(dacc(c).a)_L & \text{otherwise}\end{cases}\\
&\equiv cdata(c) = \begin{cases} \ell(c.m)(ea(c).l)_H & ea(c).[2]\\ \ell(c.m)(ea(c).l)_L & \text{otherwise}\end{cases}\\
&\equiv cdata(c) = c.m_4(ea(c))\\
&\equiv castest(c).
\end{aligned}
$$

Thus

$$M'(a) = \begin{cases} modify(\ell(c.m)(a), dmin(c), bw(c)) & write(c) \wedge a = ea(c).l \\ \ell(c.m)(a) & \text{otherwise.} \end{cases} \tag{2}$$

With $x \in \mathbb{B}^{32}$ such that

$$\begin{aligned} x.l &= a \geq 2^r \\ \langle x.o \rangle &= j \in \mathbb{B}_3 \end{aligned}$$

and the definition of function *modify*, we rewrite equation 2 using lemma 5 and lemma 6.3 from [KMP14] as follows:

$$\begin{aligned} byte(\langle x.o \rangle, M'(x.l)) &= \begin{cases} byte(j, dmin(c)) & write(c) \wedge a = ea(c).l \wedge bw(c)[j] \\ byte(j, \ell(c.m)(a)) & \text{otherwise} \end{cases} \\ &= \begin{cases} byte(i, B(c)) & write(c) \wedge a = ea(c).l \wedge \\ & j = \langle ea(c).o \rangle + i \wedge i < d(c) \\ byte(j, \ell(c.m)(a)) & \text{otherwise} \end{cases} \\ &= \begin{cases} byte(i, B(c)) & write(c) \wedge x = ea(c) +_{32} i_{32} \wedge i < d(c) \\ c.m(x) & \text{otherwise} \end{cases} \\ &= c'.m(x). \end{aligned}$$

From the result above we conclude

$$M' = \ell(c'.m).$$ □

Analogous to above we rephrase the fetch of the ISA instruction

$$I(c) = c.m_4(c.pc)$$

in terms of instruction access $iacc(c)$ to the line addressable version $\ell(c.m)$ of the ISA memory.

$$\begin{aligned} iacc(c).a &= c.pc.l \\ iacc(c).r &= 1 \end{aligned}$$

As the output to the instruction access we have

$$imout(c) = dataout(\ell(c.m), iacc(c)).$$

By the software condition we know that the program counter is word-aligned.

$$c.pc[1:0] = 00$$

Similarly to lemma 6, using properties of the memory embedding, for the output to the instruction access we conclude the following.

Lemma 7.

$$I(c) = \begin{cases} imout(c)_L & c.pc[2] = 0 \\ imout(c)_H & c.pc[2] = 1 \end{cases}$$

2.3 Interrupt Mechanism

We basically use the interrupt mechanism from [PBLS16] and [MP00]. In the specification we adjust things in several ways:

- besides *reset*, there is only one external interrupt signal $e \in \mathbb{B}$. This signal will later be generated by an advanced programmable interrupt controller (APIC). It gets priority 1 and is masked by bit 1 of the status register.

- there are two kinds of misalignment interrupts:
  i) *malf*: misaligned instruction address (*ia*) during fetch, and
  ii) *malm*: misaligned effective address (*ea*) during memory operation.
- overflow is not maskable. As instructions *add*, *addi*, and *sub* create overflow interrupts and their 'unsigned' counterparts don't; there is no need to mask twice.
- we distinguish between page faults and general-protection faults. Until we treat address translation we tie the corresponding interrupt event signals to zero.

Because misaligned memory accesses are now signaled by interrupts, we can drop the corresponding software condition. In correctness proofs, even for the sequential implementation of Chap. 7, this comes at the price of extra bookkeeping, because with a misaligned instruction fetch, signal $I(c)$ and the control signals derived from it all have no meaning; hence the proof has in this case to work without referring to them.

Implementation Details

The pipelining and forwarding of signals (see Sect. 8.1) follows in a fairly straightforward way the lines of [MP00]. But when it comes to rolling back interrupted instructions things get involved. As in [Krö01] we treat interrupts as a special case of misspeculation (we speculate that no interrupts occur and roll instructions in the pipeline back in case we discover that we were wrong), and as in [Bey05] and [BJK$^+$03] we stabilize the inputs to the cache memory system in case an instruction is rolled-back while a memory access is still in progress. But in later constructions we will have rollback due to interrupts *and* due to 'ordinary' speculative execution. For this purpose we present a new stall engine (see Sect. 8.1.1) which allows to trigger rollbacks in any pipeline stage and simultaneously stabilizes accesses to the cache memory system during such rollbacks.

2.3.1 Special Purpose Registers Revisited

The machines specified and constructed before have already a special purpose register file *c.spr*, and data can be transported between this file and the general purpose register by means of move instructions. So far however, only the *cas* instruction made use of this file by taking the 'compare data' $cdata(c)$ from register $c.spr(9_5)$. The interrupt mechanism to be introduced in this chapter makes use of quite a few more special purpose registers. In Table 8 we introduce shorthands for the first 10 registers of the special purpose register file; except for *pto*, which is used for address translation, the interrupt mechanism and the CAS instruction together make use of all of them. Thus we abbreviate

$$c.sr = c.spr(0_5)$$
$$\vdots$$
$$c.cdata = c.spr(9_5).$$

For convenience, we introduce the following helper function to map the synonyms of the special purpose registers back to the addresses of the corresponding registers in the SPR file.

$$spr(a) = Z \;\rightarrow\; \texttt{spr[}Z\texttt{]} = a$$

Also we introduce the following abbreviations for addresses of the SPR registers.

sr	esr	eca	epc	edpc	edata	pto	mode	emode	cdata
0_5	1_5	2_5	3_5	4_5	5_5	6_5	7_5	8_5	9_5

The mode register $c.mode \in \mathbb{B}^{32}$ distinguishes between *system mode*, where

$$c.mode[0] = 0,$$

Table 8: Special purpose registers

$\langle a \rangle$	synonym for $spr(a)$	name
0	sr	status register
1	esr	exception status register
2	eca	exception cause
3	epc	exception pc
4	$edpc$	exception dpc
5	$edata$	exception data
6	pto	page table origin
7	$mode$	mode register
8	$emode$	exception mode register
9	$cdata$	compare data

and *user mode*, where

$$c.mode[0] = 1.$$

We abbreviate

$$mode(c) = c.mode[0].$$

Clearing bit $sr[1]$ of the status register will mask external interrupts. After reset the machine should be in system mode, the external interrupts should be masked, and the lowest bit of the exception cause register should be set.

$$\begin{aligned} mode(c^0) &= 0 \\ c^0.sr[1] &= 0 \\ c^0.eca[0] &= 1 \end{aligned}$$

The purpose of the other special purpose registers will be explained shortly.

2.3.2 Types of Interrupts

Interrupts are triggered by interrupt event signals; they change the control flow of programs. Here we consider event signals $ev[0:10]$ from Table 9. We classify interrupts in three ways:

- internal or external. The first two interrupts are generated outside of CPU and the memory system. Renaming
  $$reset = ev[0]$$
  and
  $$e = ev[1]$$
  we collect them into the vector $eev \in \mathbb{B}^2$ of external interrupt event signals.
  $$eev[0:1] = (reset, e)$$
  The other interrupts are generated within CPU and memory system. Their activation depends only on the current ISA configuration c. We collect them into the vector $iev(c) \in \mathbb{B}^9$ of internal interrupts.
  $$iev(c)[2:10] = ev[2:10].$$
  More internal interrupt signals must be introduced for machines with floating point units (see [MP00]). In multi-core processors one tends to additionally implement inter processor interrupts.
- maskability. Only e is maskable. When an interrupt is masked, the processor will not react to it if the corresponding event signal is raised.

Table 9: Interrupts handled by the ISA

index j	synonym for $ev[j]$	name	maskable	resume
0	*reset*	reset	no	abort
1	*e*	external event signal	yes	repeat
2	*malf*	misalignment on fetch	no	abort
3	*pff*	page fault on fetch	no	repeat
4	*gff*	general-protection fault on fetch	no	abort
5	*ill*	illegal instruction	no	abort
6	*sysc*	system call	no	cont.
7	*ovf*	arithmetic overflow	no	cont.
8	*malm*	misalignment on memory operation	no	abort
9	*pfm*	page fault on memory operation	no	repeat
10	*gfm*	general-protection fault on memory operation	no	abort

- resume type: whether and where execution of a program should be resumed after handling of an interrupt. In many cases, when interrupts signal error conditions the program is simply aborted. If page faults are handled transparently one obviously wants to repeat the interrupted instruction after the missing page has been swapped into memory. For the external interrupts other than *reset* one also repeats the interrupted instruction for slightly less obvious reasons: interrupts will be handled according to their priority with small numbers signaling high priority. If an external interrupt occurs simultaneously with an internal interrupt and we would resume execution behind the interrupted instruction, then the internal interrupt (with the lower priority) would be lost. In the remaining cases one wishes to continue execution of the program behind the interrupted instruction. Clearly this should be the case for system calls, i.e., interrupts generated by the *sysc* instruction.

For the time being we have neither devices generating external interrupts nor memory management units generating page faults and general-protection faults. Therefore we will temporarily tie the corresponding event signals to zero.

2.3.3 MIPS ISA with Interrupts

Recall that the transition function δ_{isa} for MIPS ISA defined so far computes a new MIPS configuration c' from an old configuration c and the value of the reset signal.

$$c' = \delta_{isa}(c, reset)$$

We rename this transition function to δ_{isa}^{old} and the configuration computed by it to c^*.

$$c^* = \delta_{isa}^{old}(c, reset)$$

Then we define the new transition function

$$c' = \delta_M(c, eev)$$

which takes as inputs the old configuration c and the vector eev of the external event signals.

We proceed to define a predicate $jisr(c, eev)$ which indicates that a jump to the interrupt service routine is to be performed, and a predicate $eret(c, eev)$ which indicates a return from the interrupt service routine. If these signals are inactive, the machine should behave as it did before with an inactive reset signal.

$$/jisr(c, eev) \wedge /eret(c, eev) \;\rightarrow\; c' = c^*$$

For the computation of the $jisr$ predicate we unify notation for external and internal interrupts and define a vector $ca(c, eev)[0:10]$ of cause signals by

$$ca(c,eev)[i] = \begin{cases} eev[i] & i < 2 \\ iev(c)[i] & i \geq 2. \end{cases}$$

For the only maskable interrupt with index 1 we use bit 1 of the status register *sr* as a mask for this interrupt. Thus we define the vector $mca(c,eev)[0:10]$ of masked cause signals as

$$mca(c,eev)[i] = \begin{cases} ca(c,eev)[i] \wedge c.sr[i] & i = 1 \\ ca(c,eev)[i] & \text{otherwise.} \end{cases}$$

With $e = eev[1]$ and

$$mask(c) = c.sr[1]$$

the masked cause bit for the external interrupt can be redefined as

$$mca(c,eev)[1] = e \wedge mask(c).$$

We jump to the interrupt service routine if any of the masked cause bits is on:

$$jisr(c,eev) = \bigvee_i mca(c,eev)[i].$$

In cases this signal is active we define the interrupt level $il(c,eev)$ as the smallest index of an active masked cause bit (see p. 64).

$$il(c,eev) = \begin{cases} \min\{i \mid mca(c,eev)[i]\} & mca(c,eev) \neq 0_{11} \\ +\infty & \text{otherwise} \end{cases}$$

Execution of the interrupted instruction continues if the interrupt of the minimal level has type continue.

$$cont(c,eev) \equiv il(c,eev) \in \{6,7\}$$

We handle interrupts with small indices with higher priority than interrupts with high indices. Thus the interrupt level gives the index of the interrupt that will receive service. With an active $jisr(c,eev)$ signal many things happen at the transition to the next state c':

- we jump to the start addresses *sisr* and $sisr +_{32} 4_{32}$ of the interrupt service routine. We fix *sisr* to 0_{32}, i.e., to the first address in the ROM.

$$c'.dpc = 0_{32}$$
$$c'.pc = 4_{32}$$

- the maskable interrupt event signal *e* is masked.

$$c'.sr = 0_{32}$$

  The purpose of this mechanism is to make the calling of the interrupt handler (which will take some instructions) not interruptible by external interrupts. Of course, the interrupt service routine should also be programmed in such a way that its execution does not produce internal interrupts.
- the old value *c.sr* of the status register is saved into the exception status register *c.esr*.

$$c'.esr = c.sr$$

  Thus it can be restored later. This does not work, if the interrupted instruction writes the status register (*movs2g*) and the resume type is continue. But only system calls and arithmetic instructions have resume type continue, and they don't write the special purpose register file.

- in the exception registers *epc* and *edpc* we save something similar to the link address of a function call. It is the pair of addresses where program execution will resume after the handling of the interrupt, if it is not aborted. In ISA (and the hardware) we only distinguish if the resume type is continue or not. In all other cases we prepare for repetition of the interrupted instruction by saving the current *pc* and *dpc*. No harm is done by this, if the handler/operating system decides to abort execution.

$$c'.(edpc, epc) = \begin{cases} c^*.(dpc, pc) & cont(c, eev) \\ c.(dpc, pc) & \text{otherwise} \end{cases}$$

- in the exception data register *edata*, we save the effective address $ea(c)$. After a page fault on memory operation this provides the effective address, which generated the interrupt to the page fault handler.

$$c'.edata = ea(c)$$

- in the exception cause register *eca*, we save the interrupt level in the one-hot encoding. To produce the value saved we use the first-one circuit from [KMP14].[2]

$$c'.eca = 0_{21} \circ f1(mca(c, eev))$$

- we back-up the current machine mode into the exception mode register.

$$c'.emode = c.mode$$

- finally, we switch to system mode.

$$c'.mode = 0_{32}$$

- in case of continue interrupts we need to finish the interrupted instruction. We collect the special purpose registers that are updated at *jisr* into set

$$J = \{sr, esr, eca, epc, edpc, edata, mode, emode\}.$$

  Then for the general purpose register file and the memory we define

$$c'.(gpr, m) = \begin{cases} c^*.(gpr, m) & cont(c, eev) \\ c.(gpr, m) & \text{otherwise} \end{cases}$$

  and for special purpose registers $spr(x) \notin J$ we specify

$$c'.spr(x) = c.spr(x).$$

  Note, no harm is done by the latter 'simple' specification, because instructions generating continue interrupts do not update the special purpose register file.

This completes the definition of what happens on activation of the $jisr(c, eev)$ signal.

During a return from exception, i.e., if predicate $eret(c, eev)$ is active, also several things happen simultaneously:

- *pc* and *dpc* are restored resp. from the exception *pc* and exception *dpc*.

$$c'.(dpc, pc) = c.(edpc, epc)$$

- the status register is restored from the exception status register.

$$c'.sr = c.esr$$

- the mode register is restored from the exception mode register.

$$c'.mode = c.emode$$

[2] Specification of the first-one circuit is as follows. For $x \in \mathbb{B}^n$ we define $f1(x) \in \mathbb{B}^n$ s.t.

$$f1(x)[i] \leftrightarrow x \neq 0_n \wedge i = \min\{k \mid x[k]\}.$$

2.3.4 Specification of Most Internal Interrupt Event Signals

Except for the fault event signals, which obviously depend on the not yet defined mechanism of address translation, we already can specify when internal event signals are to be activated.

- illegal instruction. By inspection of the tables in Sect. 2.1.1 we define a predicate $undefined(c)$ which is true if the current instruction $I(c)$ is not defined in the tables. Moreover we forbid in user mode i) the access of the special purpose registers by move instructions, except for $movg2s$ instructions with the target register number 9 ($c.cdata$)[3], as well as ii) the execution of the $eret$ instruction. Moreover we forbid in any mode explicit moves to the mode register. Thus $mode$ can only be changed by interrupts and $eret$.

$$\begin{aligned} ill(c) \equiv\ & undefined(c) \vee \\ & mode(c) \wedge eret(c) \vee \\ & mode(c) \wedge move(c) \wedge /(movg2s(c) \wedge xad(c) = 8_5) \vee \\ & movg2s(c) \wedge xad(c) = 7_5 \end{aligned}$$

- misalignment. Misalignment occurs during fetch, if the low order bits of the pc are not both zero. It also occurs during a memory operation, if the effective address $ea(c)$ interpreted as a binary number is not a multiple of the access width $d(c)$.

$$\begin{aligned} malf(c) &\equiv c.dpc[1:0] \neq 00 \\ malm(c) &\equiv mop(c) \wedge (d(c) \nmid \langle ea(c) \rangle) \end{aligned}$$

- system call. This event signal is simply identical with the predicate decoding a system call instruction.

$$sysc(c) \ \equiv\ opc(c) = 0^6 \wedge fun(c) = 001100$$

- arithmetic overflow. This signal is taken from the overflow output of the ALU specification if the ALU is used.

$$ovf(c) \ \equiv\ alu(c) \wedge alu.ovf(c)$$

- page faults and general-protection faults. For the time being we tie the corresponding event signals to zero.

Except for the generation of the fault signals this already completes the formal specification of the interrupt mechanism. Note, computation of most of the interrupt event signals above is trivial. In order to justify computation of the misalignment on memory operation, we proceed to show the following technical result.

Lemma 8. *Assume* $d(c) > 0$.

$$d(c) \nmid \langle ea(c) \rangle \ \leftrightarrow\ mmask(c)[2:1] \wedge ea(c)[1:0] \neq 0_2$$

Proof of lemma 8. For convenience in the scope of this proof we use the following abbreviations.

$$\begin{aligned} d &= d(c) \in \{1,2,4\} \\ ea &= ea(c) \in \mathbb{B}^{32} \\ mm &= mmask(c) = 0^{4-d} \circ 1^d \end{aligned}$$

First, for $k = \log d$ we argue that access width d divides effective address ea if and only if the first k low order bits of ea are zeros:

$$\begin{aligned} d \mid \langle ea \rangle &\leftrightarrow \exists z \in \mathbb{Z}:\ \langle ea \rangle = z \cdot d \\ &\leftrightarrow \exists z \in \mathbb{Z}:\ \langle ea \rangle = z \cdot 2^k + 0 \\ &\leftrightarrow \exists z \in \mathbb{Z}:\ \langle ea[31:k] \rangle = z \ \wedge\ \langle ea[k-1:0] \rangle = 0 \qquad \text{(lemma 1)} \\ &\leftrightarrow ea[k-1:0] = 0_k \end{aligned}$$

[3] This move to the special purpose register is allowed in user mode in order to let unprivileged users set the compare data for the CAS instruction appropriately.

and therefore
$$d \nmid \langle ea \rangle \ \leftrightarrow\ 0^{2-k} \circ ea[k-1:0] \neq 0_2.$$

For memory mask *mm* we proceed to show the following.

$$\begin{aligned} mm[j] &\leftrightarrow (0^{4-d} \circ 1^d)[j] \\ &\leftrightarrow d > j \\ &\leftrightarrow k > \log j \\ &\leftrightarrow (0^{2-k} \circ 1^k)[\log j] \end{aligned} \tag{3}$$

Finally, using the result above we derive

$$\begin{aligned} 0^{2-k} \circ ea[k-1:0] &\leftrightarrow (0^{2-k} \wedge ea[1:k]) \circ (1^k \wedge ea[k-1:0]) \\ &\leftrightarrow (0^{2-k} \circ 1^k) \wedge ea[1:0] \\ &\leftrightarrow mm[2:1] \wedge ea[1:0] \qquad \text{(equation 3)} \end{aligned}$$

and the claim follows. □

2.3.5 Accesses of ISA Revisited

Due to misalignment on fetch an instruction accesses can now be void, and we redefine its read component as

$$iacc(c).r = \begin{cases} 0 & malf(c) \\ 1 & \text{otherwise.} \end{cases}$$

Similarly data accesses can become void due to interrupts, which are not of type continue; note that this includes misalignment on memory operation. We redefine the read, write, and CAS components of these accesses as

$$dacc(c).(r,w,cas) = \begin{cases} (0,0,0) & jisr(c,eev) \wedge /cont(c,eev)(c) \\ (l(c), s(c), cas(c)) & \text{otherwise.} \end{cases}$$

Lemma 6 relating the data access of ISA with the memory update stays literally the same. In the proof interrupts which are not of type continue now have to be treated as an additional case.

Lemma 9.
$$\ell(c'.m) = \delta_M(\ell(c.m), dacc(c))$$

2.4 Multi-Level Address Translation

This is the last section literally taken from [LOP]. It serves a basis for development of a more advanced address translation mechanism in Chap. 3.

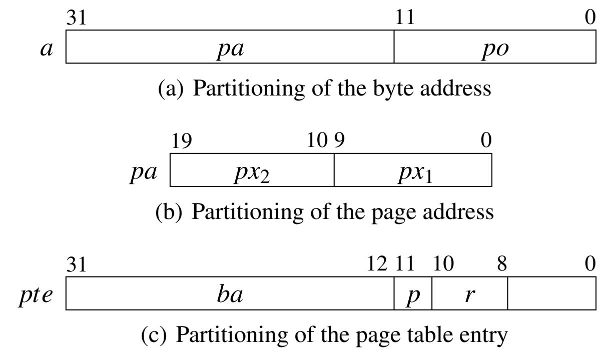
2.4.1 Page Tables

Pages have $4096 = 4K$ bytes reps. $1024 = K$ words. As illustrated in Fig. 6(a), byte addresses $a \in \mathbb{B}^{32}$ are partitioned into page addresses and page offsets:

$$\begin{aligned} a.pa &= a[31:12] \\ a.po &= a[11:0]. \end{aligned}$$

For $i \in [2:1]$, page addresses $pa \in \mathbb{B}^{20}$ are further partitioned into level i page indices $pa.px_i$ as illustrated in Fig. 6(b) by

$$\begin{aligned} pa.px_2 &= pa[19:10] \\ pa.px_1 &= pa[9:0]. \end{aligned}$$



(a) Partitioning of the byte address

(b) Partitioning of the page address

(c) Partitioning of the page table entry

Fig. 6: Partitioning of address and page table entries

Page table entries $pte \in \mathbb{B}^{32}$ are one word long, thus K of them fit on a single page. A page whose words are used as page table entries is called a *page table*. Page table entries are partitioned into base address $pte.ba \in \mathbb{B}^{20}$, present bit $pte.p \in \mathbb{B}$, and rights bits $pte.r \in \mathbb{B}^3$ as indicated in Fig. 6(c) by

$$\begin{aligned} pte.ba &= pte[31:12] \\ pte.p &= pte[11] \\ pte.r &= pte[10:8]. \end{aligned}$$

The bits $r[2:0]$ of a rights vector for a page will be interpreted in the following way:

- $r[0] = r.w$: write permission,
- $r[1] = r.u$: permission to access as a user, and
- $r[2] = r.ex$: permission to fetch as an instruction and execute.

2.4.2 Walks and Translation Requests

Intuitively, multilevel (here: 2-level) address translation is achieved by walking a graph of page tables, whose edges are defined by the base address fields of the page table entries. In each level i of translation, the edge to follow from a page table is determined by the level i page index $a.px_i$ of the page address a which is translated. The central concept for formalizing this are walks w, which have the following components:

- $w.a \in \mathbb{B}^{20}$: the page address to be translated.
- $w.\ell \in \{100, 010, 001\}$: one-hot encoding of the number of walk extensions still required.
- $w.ba \in \mathbb{B}^{20}$: the base address of the page table from which the walking is to be continued.
- $w.r \in \mathbb{B}^3$: the rights still remaining.
- $w.f \in \mathbb{B}$: the fault bit indicating that the page table entry, from which the walk was extended, was not present.

The set of all walks is denoted by

$$K_{walk} \subseteq \mathbb{B}^{47}$$

and the set of all non-faulting walks is denoted by

$$K_{walk}^{+} = \{w \in K_{walk} \mid w.f = 0\}.$$

Walks can be created and extended. For the initiation of a walk one needs a page address $a \in \mathbb{B}^{20}$ to be translated and a page table origin $pto \in \mathbb{B}^{32}$, which will come from a special purpose register file. The initial walk

$$w = winit(a, pto)$$

has the following components:

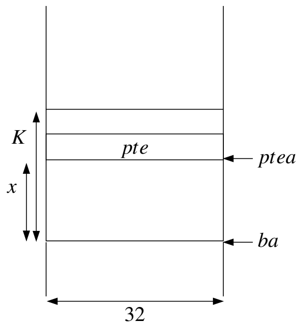


Fig. 7: Page table entry address

- $w.a = a$. The page address to be translated.
- $w.\ell = 100$. All levels of translation still need to be performed.
- $w.ba = pto.pa$. Translation starts at the page table with base address $pto.pa$.
- $w.r = 111$. Rights have not yet been restricted.
- $w.f = 0$. No missing page table entry has been found, because no page table entry has yet been accessed.

We assume that the page table origin pto is page aligned, i.e., that

$$pto[11:0] = 0^{12}.$$

For a base address $ba \in \mathbb{B}^{20}$ and a page index $x \in \mathbb{B}^{10}$ we define the page table entry address $ptea(ba,x)$, i.e., the address of the page table entry on page ba with index x as

$$\begin{aligned} ptea(ba,x) &= ba \circ 0^{12} +_{32} 0^{20} \circ x \circ 00 \\ &= ba \circ x \circ 00. \end{aligned}$$

This is illustrated in Fig. 7. Overloading notation we define the page table entry address accessed for extending walk w as

$$ptea(w) = ptea(w.ba, w.a.px_{level(w)}).$$

Thus the page table entry address is determined by the base address $w.ba$ of the walk and the page index $w.a.px_i$ of the address under translation, where $i = level(w)$ is the level of the walk (which determines the number of walk extensions still required).

$$level(w) = \begin{cases} 2 & w.\ell = 100 \\ 1 & w.\ell = 010 \\ 0 & w.\ell = 001 \end{cases}$$

Walks with level 0 are called complete and cannot be further extended.

$$complete(w) \equiv level(w) = 0$$

For walk extension with a memory m one looks up the page table entry $pte(w,m)$ in memory m at address $ptea(w)$, i.e.,

$$pte(w,m) = m_4(ptea(w)).$$

The extension

$$w' = wext(w,m)$$

of incomplete walk w with memory m is defined as follows.

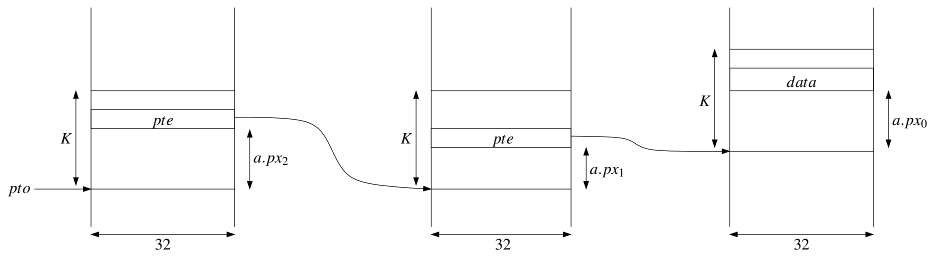


Fig. 8: Process of address translation

$$\begin{aligned} w'.a &= w.a \\ w'.f &= /pte(w,m).p \\ w'.\ell &= \begin{cases} w.\ell & w'.f \\ 0 \circ w.\ell[2:1] & /w'.f \end{cases} \\ w'.ba &= pte(w,m).ba \\ w'.r &= pte(w,m).r \wedge w.r \end{aligned}$$

A few simple properties of walks are listed in the following lemma.

Lemma 10.

- *Non faulting walk extension decreases the level.*

$$/w'.f \rightarrow level(w') = level(w) - 1$$

- *Complete walks are not faulting.*

$$complete(w) \rightarrow /w.f$$

For virtual addresses $va \in \mathbb{B}^{32}$ and complete walks w with page address

$$w.a = va.pa$$

we define the translated memory address $tma(va, w)$ obtained by translating va with walk w as

$$complete(w) \wedge w.a = va.pa \;\rightarrow\; tma(va, w) = w.ba \circ va.po.$$

The process of address translation by repeated walk extension is illustrated in Fig. 8. Before we proceed, for bit strings x and y of equal length we define:

$$x \leq y \;\equiv\; \forall i : x_i \leq y_i.$$

A translation request trq has two components:

- $trq.a \in \mathbb{B}^{20}$: the page address to be translated, and
- $trq.r \in \mathbb{B}^{3}$: the access rights requested.

A walk w matches a translation request trq if its address $w.a$ equals $trq.a$ and if w is faulting or complete.

$$match(trq, w) \equiv trq.a = w.a \wedge (w.f \vee complete(w))$$

A matching walk provides a translation for the request if the walk is complete and provides at least the rights requested.

$$trans(trq, w) \equiv match(trq, w) \wedge /w.f \wedge (trq.r \leq w.r)$$

A matching walk leads to a page fault for the request if it is faulting.

$$pfault(trq, w) \equiv match(trq, w) \wedge w.f$$

It leads to a general-protection fault if it is complete and the rights are insufficient.

$$gfault(trq, w) \equiv match(trq, w) \wedge /w.f \wedge /(trq.r \leq w.r))$$

2.4.3 MIPS ISA with Address Translation

Due to address translation the instruction fetch stage will be preceded in the pipelined implementation by a stage for the translation of the instruction address. This gives rise to a second delay slot. Thus we need 3 program counters pc, dpc, and $ddpc$ and their counter parts epc, $edpc$, and $eddpc$ in the special purpose register file.

$$c'.pc = \begin{cases} 8_{32} & jisr(c,eev) \\ c.epc & eret(c) \wedge /jisr(c,eev) \\ nextpc(c) & \text{otherwise} \end{cases}$$

$$c'.dpc = \begin{cases} 4_{32} & jisr(c,eev) \\ c.edpc & eret(c) \wedge /jisr(c,eev) \\ c.pc & \text{otherwise} \end{cases}$$

$$c'.ddpc = \begin{cases} 0_{32} & jisr(c,eev) \\ c.eddpc & eret(c) \wedge /jisr(c,eev) \\ c.dpc & \text{otherwise} \end{cases}$$

In ISA, the instruction address is

$$ia(c) = c.ddpc$$

which will subsequently be translated or not depending on the current mode $mode(c)$.

Configurations $c \in K_{M+T}$ are now triples with the components:

- $c.core = c.core.(pc, dpc, ddpc, gpr, spr)$: processor core,
- $c.m : \mathbb{B}^{32} \rightarrow \mathbb{B}^{8}$: byte addressable memory, and
- $c.tlb \in K_{tlb}$: a TLB containing the walks currently available for address translation or extension.

A translation look-aside buffer (TLB) is a cache for translations. Formally we simply define it as a set of walks. Thus, if K_{walk} is the set of configurations of walks, then the set K_{tlb} of TLB configurations is

$$K_{tlb} = 2^{K_{walk}}.$$

The model allows the presence of different translations for the same page address in the TLB. ISA for MIPS with TLB is nondeterministic in four respects:

- the interleaving of processor core steps and TLB steps,
- the (speculative) choice of initial walks to be placed in the TLB,
- the choice of a walk in the TLB that is to be extended, and
- the choice of walks w_I for the translation of the pc for instruction fetch and w_E for the translation of effective addresses among the possibly multiple matching walks in the TLB.

Nondeterminism is formalized by a transition function whose inputs have real components as well as oracle components. Formally we define an input alphabet Σ for a transition function

$$\delta_{M+T} : K_{M+T} \times \Sigma \rightarrow K_{M+T}$$

mapping configurations c and inputs x into a next configuration

$$c' = \delta_{M+T}(c, x).$$

There are two major cases: TLB steps and core steps. For core steps, we have to process the external interrupt vector eev and we have, among other things, to deal with the new instructions $flusht$ and $invlpg$ from Table 2. For the time being we do not deal with tagged TLBs, and thus we ignore ASIDs specified in the rs field of the $invlpg$ instruction.

The input alphabet Σ is the disjoint union of alphabets Σ_{core} and Σ_{tlb} for processor core and TLB steps.

$$\Sigma = \Sigma_{core} \dot{\cup} \Sigma_{tlb}$$

2.4.4 TLB Steps

Inputs leading to TLB steps are only allowed in translated mode, i.e., when

$$mode(c) = 1.$$

In system mode the ISA TLB does not walk. This gets more complicated if one builds hardware support for hypervisors.
For all page addresses $a \in \mathbb{B}^{20}$ we include the walk initialization step into the input alphabet.

$$(winit, a) \in \Sigma_{tlb}$$

This step has the effect of placing an initial walk for a and page table origin $c.pto$ into the TLB. The processor core configuration is not changed.

$$c'.tlb = c.tlb \cup \{initw(a, c.pto)\}$$
$$c'.core = c.core$$

For all walks $w \in K_{walk}$ we also include the walk extension step into the input alphabet.

$$(wext, w) \in \Sigma_{tlb}$$

The next state is only defined for incomplete walks in the TLB.

$$/complete(w) \wedge w \in c.tlb$$

For such walks w the extension of w with memory $c.m$ is included into the TLB. The processor core configuration stays unchanged.

$$c'.tlb = c.tlb \cup \{wext(w, c.m)\}$$
$$c'.core = c.core$$

Note, in case of a faulty walk extension, the resulting faulty walk is stored in the TLB. The latter walk is used to signal a fault on the forthcoming processor core step.

2.4.5 Processor Core Steps

For components $x \in \{pc, dpc, ddpc, gpr, spr\}$ we abbreviate by overloading notation

$$c.x = c.core.x.$$

We denote undefined walks by the symbol $\perp$, and for walks

$$w_I, w_E \in K_{walk} \cup \{\perp\}$$

and all external interrupt vectors

$$eev[1:0] = (e, reset) \in \mathbb{B}^2$$

we include into Σ_{core} the input symbol

$$x = (w_I, w_E, eev) \in \Sigma_{core}.$$

Walks which are used are required to be in the TLB. For such inputs the machine performs a processor core step. In translated mode, walk w_I is used for translation of the instruction address and walk w_E is used for translation of the effective address in memory operations. The formal definition of c' is necessarily somewhat lengthy, because we have to adapt the definition of the internal interrupt event signals to the case where the machine might be running in translated mode. The interesting part of the definitions concerns of course the case when the machine is running in translated mode, i.e., when

$$mode(c) = mode(c.core) = 1.$$

Reset

For inputs

$$x = (w_I, w_E, e1)$$

we flush the TLB, initialize the program counters and put the processor into the system mode:

$$\begin{aligned} c'.tlb &= \emptyset \\ c'.(ddpc, dpc, pc) &= (0_{32}, 4_{32}, 8_{32}) \\ c'.mode &= 0_{32}. \end{aligned}$$

Misalignment on Fetch

This is not affected by address translation.

$$malf(c) = malf(c.core)$$

Faults on Fetch

The translation request for instruction fetch is

$$trq_I(c) = (ia(c).pa, 110).$$

If the machine is running in translated mode, we require walk w_I to be a walk from the TLB matching translation request $trq_I(c)$.

$$mode(c) \;\rightarrow\; match(trq_I(c), w_I) \wedge w_I \in c.tlb$$

We get a page fault or general-protection fault on fetch if translation of this request with the chosen walk w_I produces the corresponding fault.

$$\begin{aligned} pff(c,x) &\equiv mode(c) \wedge pfault(trq_I(c), w_I) \\ gff(c,x) &\equiv mode(c) \wedge gfault(trq_I(c), w_I) \end{aligned}$$

For convenience we abbreviate:

$$ff(c,x) = pff(c,x) \vee gff(c,x).$$

Instruction Fetch

Instruction fetch is only possible in the absence of misalignment and — in translated mode — the absence of page fault and general-protection fault on fetch. Then we can define the physical memory address pma_I for instruction memory access. In system mode it is $ia(c)$. In user mode $ia(c)$ is translated with walk w_I.

$$/malf(c) \wedge /ff(c,x) \;\rightarrow\; pma_I(c,x) = \begin{cases} ia(c) & mode(c) = 0 \\ tma(ia(c), w_I) & mode(c) = 1 \end{cases}$$

The instruction to be executed is then fetched from pma_I.

$$I(c,x) = c.m_4(pma_I(c,x))$$

Now we write all functions $f(c)$ which depend only on the instruction $I(c)$ as

$$f(c) = f'(I(c))$$

and generalize them to

$$f(c,x) = f'(I(c,x)).$$

This covers function fields like *rd*, predicates like *addi* or *ls*, addresses like *xad*, and also some interrupt event signals like *sysc*. For any such function f we write the old definition of $f(c)$ as a function f' of $I(c)$. Substituting in subsequent definitions of functions $F(c, eev)$ these functions $f(c)$ by functions $f(c,x)$ we obtain properly generalized versions $F(c,x)$ of these functions as long as memory or new instructions are not involved. For instance:

$$\begin{aligned} A(c,x) &= c.gpr(rs(c,x)) \\ ea(c,x) &= A(c,x) +_{32} sxtimm(c,x). \end{aligned}$$

Illegal Interrupt

In user mode instructions *flush* and *invlpg* are also illegal.

$$\begin{aligned} ill(c,x) \equiv\ & undefined(c,x) \vee \\ & mode(c) \wedge eret(c,x) \vee \\ & mode(c) \wedge (flush(c,x) \vee invlpg(c,x)) \vee \\ & mode(c) \wedge move(c,x) \wedge /(movg2s(c,x) \wedge xad(c,x) = 8_5) \end{aligned}$$

Overflow and Misalignment on Memory Operation

The two definitions below are repeated for completeness of presentation.

$$\begin{aligned} ovf(c,x) &\equiv alu(c,x) \wedge alu.ovf(c,x) \\ malm(c,x) &\equiv mop(c,x) \wedge (d(c,x) \nmid \langle ea(c,x) \rangle) \end{aligned}$$

Faults on Memory Operation

The translation request for the effective address is

$$trq_E(c,x) = (ea(c,x).pa, 01 \circ (s(c,x) \vee cas(c,x))).$$

If the machine is executing a memory operation in translated mode we require walk w_E to be a walk from the TLB matching translation request $trq_E(c,x)$.

$$mode(c) \wedge mop(c,x) \ \rightarrow \ match(trq_E(c,x), w_E) \wedge w_E \in c.tlb$$

We get a page fault or general-protection fault on memory operation if translation of this request with the chosen walk w_E produces a fault.

$$\begin{aligned} pfm(c,x) &\equiv mode(c) \wedge pfault(trq_E(c,x), w_E) \\ gfm(c,x) &\equiv mode(c) \wedge gfault(trq_E(c,x), w_E) \end{aligned}$$

Analogous to above we abbreviate:

$$fm(c,x) = pfm(c,x) \vee gfm(c,x).$$

Memory Operation

Execution of a memory operation is only possible in the absence of misalignment and — in translated mode — any fault on memory operation. Then we can define the physical memory address pma_E for data memory access. In system mode it is $ea(c,x)$. In user mode $ea(c,x)$ is translated with walk w_E.

$$/malm(c,x) \wedge /fm(c,x) \ \rightarrow \ pma_E(c,x) = \begin{cases} ea(c,x) & mode(c) = 0 \\ tma(ea(c,x), w_E) & mode(c) = 1 \end{cases}$$

Next we split cases on the memory operation performed. For loads ($l(c,x)$) we have

$$\begin{aligned} fill(c,x) &= \begin{cases} 0 & u(c,x) \\ c.m(pma_E(c,x) +_{32} (d(c,x)-1)_{32})[7] & \text{otherwise} \end{cases} \\ lres(c,x) &= fill(c,x)^{32-8\cdot d(c,x)} \circ c.m_{d(c,x)}(pma_E(c,x)). \end{aligned}$$

For stores ($s(c,x)$) we have

$$c'.m(y) = \begin{cases} byte(i, c.gpr(rt(c,x))) & y = pma_E(c,x) +_{32} i_{32} \wedge i < d(c,x) \\ c.m(y) & \text{otherwise.} \end{cases}$$

Finally, in case of CAS ($cas(c,x)$), we have

$$\begin{aligned} lres(c,x) &= c.m_4(pma_E(c,x)) \\ c'.m(y) &= \begin{cases} byte(i, c.gpr(rt(c,x))) & castest(c,x) \wedge y = pma_E(c,x) +_{32} i_{32} \wedge i < 4 \\ c.m(y) & \text{otherwise.} \end{cases} \end{aligned}$$

Instruction Execution without invlpg, flush, or reset

Let

$$\delta_M : K_M \times \mathbb{B}^2 \to K_M$$

be the ISA transition function without address translation. From this we obtain

$$\delta_x : K_M \times \mathbb{B}^2 \to K_M$$

by replacing in the ISA specification all functions $f(c)$ that have been generalized or changed by $f(c,x)$. Then

$$\begin{aligned}(c'.core, c'.m) &= \delta_x(c.core, c.m, e \circ 0)\\ c'.tlb &= c.tlb.\end{aligned}$$

Flush or invlpg

For the time being we use a single address space and no address space identifiers (ASIDs). Execution of a flush instruction in system mode flushes the TLB.

$$flush(c,x) \;\to\; c'.tlb = \emptyset$$

For technical reason, the definition of intermediate result C is extended: in case of *invlpg* instructions we include $B(c).pa$ in the lower bits of the intermediate result C. This is the page index whose translation in the TLB will be invalidated by the instruction. If we would use ASIDs, we could store them in the upper 12 bits of C.

$$C(c,x) = \begin{cases} S(c,x) & movs2g(c,x)\\ B(c,x) & movg2s(c,x)\\ 0^{12} \circ B(c,x).pa & invlpg(c,x)\\ su.res(c,x) & su(c,x)\\ linkad(c,x) & jal(c,x) \vee jalr(c,x)\\ alu.res(c,x) & \text{otherwise}\end{cases}$$

Execution of an *invlpg* instruction in system mode removes from the TLB the walks translating page address given by the GPR register specified by the *rt* field from the TLB. It also removes all incomplete walks.

$$invlpg(c,x) \;\to\; c'.tlb = \{w \in c.tlb \mid (w.a \neq C(c,x)[19:0]) \wedge complete(w)\}$$

In both cases the register files and the memory are not changed and the program counters are updated.

$$\begin{aligned} Z \in \{gpr, spr, m\} \to c'.Z &= c.Z\\ c'.pc &= c.pc +_{32} 4_{32}\\ c'.dpc &= c.pc\\ c'.ddpc &= c.dpc\end{aligned}$$

Part II

Nested Address Translation (NAT)

3 Introduction and Specification

In this chapter we develop a more general scheme of address translation. This new scheme is introduced in order to enhance virtualization capabilities of our machine and gain hardware support for the so called "hypervisors", special kind of kernels which we present later. In a nutshell, we add one more layer of the virtual memory. This allows the machine to run in three different translation modes:

- untranslated — same as *system mode* before,
- translated — same as *user mode* before, and
- nested-translated — newly added.

As a result, now *user* programs can run translated. In the new translation mode, the machine fetches code and accesses memory after two phases of translation:

i) translation of the instruction or effective address to the level of user addresses, and
ii) ordinary translation (as before) of the user level address to the physical address.

An important detail, which might be hard to notice from the first glance, is that the first phase of translation itself utilizes several rounds of translation, which are essentially of the same type as in translation phase two. This overhead is simply due to the fact that one can *not* access memory directly in the first phase, since now — with an extra layer of virtual memory — page tables accessed during walking in the first phase are filled with user level (virtual) addresses. Thus, in order to get to the physical addresses, one has to perform quite a few intermediate ordinary translations, which is known as *quadratic walking* [Meg12]. How many and what kind of accesses we perform for this so called *nested translation* we describe in detail later. However, already at this point one can think about the nested translation as of more general translation process where the memory accesses in their turn are preceded by the translation calls. We try to reflect this idea in our hardware descriptions to strengthen the intuition.

Just as for any newly added mechanisms, we start with informal description to create certain background and formalize new concepts.

3.1 Virtualization

Aiming at virtualization support for hypervisors, from the variety of software and hardware techniques available for virtualization [Age09], we of course implement the virtual address translation, as the most crucial one, together with the mechanism of intercepts. Since hypervisors require two phases of address translation, whereas the hardware might provide only one phase, the second phase of translation can be implemented in software [Kov13], usually with the help of data structures called the *shadow page tables*. In contrast to [Kov13], we implement the second phase of address translation in hardware.

3.1.1 Translation Modes

The first thing we need in order to increase the number of supported translation modes is an additional

- 32-bit data field to hold the origin of one extra page table structure, and
- 1-bit control flag to keep track whether the new mode is activated.

Conveniently, we make a room for these data in the special purpose register file, and bind the registers number 11 and 12 resp. to store these additional page table origin and mode. For technical reasons (to backup the new mode on interrupts) we also require one more exception register. As usual, it is taken from the SPR, where it gets number 13. For the newly involved registers we choose the following synonyms:

$$\begin{aligned} c.npto &= c.spr(11_5) \\ c.nmode &= c.spr(12_5) \\ c.enmode &= c.spr(13_5). \end{aligned}$$

Now we have to adjust our notation from the previous chapters. There using the last bit of the mode register

$$mode(c) = c.mode[0]$$

we distinguished between the *user mode* (bit set) and the *system mode* (not set), and the *user mode* was the only translated mode. Clearly, in the prior versions of our specification it was a completely reasonable and elegant solution to keep track of the translation mode using predicate $mode(c)$. Since the number of modes increases, we can no longer use this predicate. Instead, we introduce three new predicates — one per mode — and finally fix the names. We redefine

$$mode(c) = c.nmode[0] \circ c.mode[0]$$

and abbreviate

$$\begin{aligned} user(c) &\equiv mode(c) = 11 \\ guest(c) &\equiv mode(c) = 01 \\ host(c) &\equiv mode(c) \in \{10, 00\}. \end{aligned}$$

As it might be hinted by naming, the *user mode* now stands for the newly added mode of nested translation, while the *guest mode* and the *host mode* stand for the ordinary translated mode (old *user mode*) and the untranslated mode (old *system mode*) resp.

3.1.2 Intercept Mechanism

Together with adding of translation modes we simultaneously introduce new privilege levels for program execution. In our case, the resulting number of these levels equals the number of supported translation modes, and for referring to a certain privilege level we use the name that the corresponding translation mode has. In order to control an execution at a less privileged level, at a more privileged level one should be able to specify *what is allowed to happen* without pausing/aborting (an intercept) that execution. Following this intuition we describe things formally.

Let x be the machine's input for a processor step, like defined in Chap. 2. The new semantics of *jisr* and *eret* are defined as follows. The *jisr* now can occur at any out of three privilege levels, and — depending on context — switch the machine to the privilege level of host or guest.

$$\begin{aligned} jisr(c,x) \wedge (user(c) \wedge \overline{icpt(c,x)}) &\rightarrow \quad c'.nmode[0] = 0 \\ jisr(c,x) \wedge (\overline{user(c)} \vee icpt(c,x)) &\rightarrow \quad c'.mode[0] = 0 \end{aligned}$$

Note, we specify the change of *c.mode* in a slightly redundant way: *jisr* at the level of host should not cause any changes, whereas we clear the $c.mode[0]$. That is not a problem since the $c.mode[0]$ is zero in this case anyway.

Table 10: Changes of the mode on interrupts (JISR) and returns from exceptions (ERET)

JISR	user (11)	guest (01)	host (10)	host (00)
$/icpt$	guest (01)	guest (01)	host (10)	host (00)
$icpt$	host (10)	host (00)	host (10)	host (00)

ERET	user (11)	guest (01)	host (10)	host (00)
$/expt$	—	guest (01)	host (10)	host (00)
$expt$	—	user (11)	user (11)	guest (01)

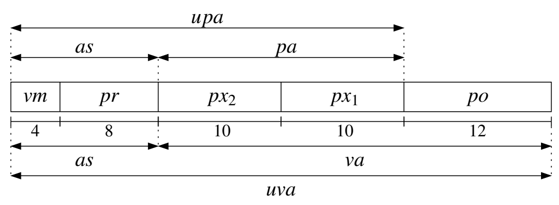


Fig. 9: Partitioning of universal addresses

On *eret*, which is now allowed at the level of guest as well, the machine normally switches back to the privileged level it was running at before the interrupt.[1]

$$eret(c,x) \wedge \overline{host(c)} \rightarrow c'.nmode = c.enmode$$
$$eret(c,x) \wedge host(c) \rightarrow c'.mode = c.emode$$

While we manage to keep most of the old specifications, we obtain a variety of new machine's behaviors. In Table 10 we collect all possible cases of the mode change. Thus, on *jisr* either we drop from the level of user or guest — depending on the intercept signal — to the level of guest or host, or the current mode stays unchanged. Note that on *jisr* at most one of the $mode(c) \in \mathbb{B}^2$ bits changes. To give the change of mode on *eret*, we introduce a predicate *exception*:

$$expt(c) \equiv \begin{cases} c.emode[0] & host(c) \\ c.enmode[0] & guest(c). \end{cases}$$

The *eret* instruction — depending on the current mode and the exception — brings us to the level of guest or user, or the current mode stays unchanged. An interesting detail is that from the level of user we might drop directly to the level of host on *jisr*, and importantly, return directly to the level of user with *eret*.

For the privilege levels of user and guest we specify the signal $icpt(c,x)$ in Sect. 3.3.4. At the most privileged level of host any computational behavior is allowed, and therefore nothing is intercepted.

$$host(c) \rightarrow icpt(c,x) = 0$$

3.1.3 Universal Addressing

In a complex computer system, where multiple users are running under control of their operating system(s), virtual memory is a key technique for separation and isolation of accesses to the single and conceptually shared computer's memory. Also the virtual memory — combined with mechanisms of paging and swap memory — allows to access a memory larger

[1] On the *eret*, the machine is not necessarily switches to the previous mode (the mode where an interrupt had occurred). Recall, now the mode is determined by a combination of values coming from two different registers. On return from exception one of these registers is simply restored from the corresponding exception register. This, of course, does not mean that value of the latter register was preserved during the interrupt handing.

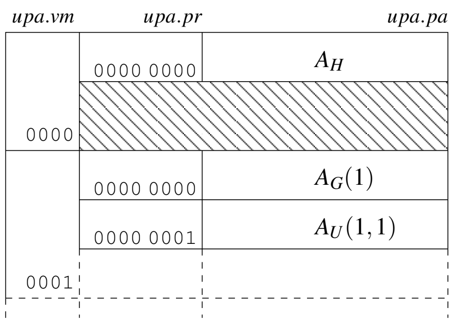


Fig. 10: Address space of the universal page address *upa*

that the physically installed one [Hil05]. Normally, user programs running under the memory virtualization cannot communicate — unless a special effort is put to implement certain data exchange mechanisms — and each program sees the memory as huge as it can address using the provided ISA.

Since the user addresses are *not* unique by their nature, they are subject for address translation, the result of which depends on the unique address space ID (ASID) *as* under which the program is running. This ID consists of the virtual machine ID *as.vm* (4 bits) and the process ID *as.pr* (8 bits), which are extracted directly from the processor's core. The virtual byte address together with the ASID form a *universal virtual address*, which we typically denote by $uva \in \mathbb{B}^{44}$. Since the address translation mostly operates with the page addresses, we introduce the *universal page addresses* as well. Obviously, these addresses are compositions of the ASIDs and the regular page address, and typically we denote these addresses by $upa \in \mathbb{B}^{32}$. For convenience we partition the universal addresses in many different ways as depicted in Fig. 9. The corresponding definitions are obvious, and therefore are omitted.

As a convention, we reserve the virtual machine ID (VMID) consisting of all zeros for programs which run at the privilege level of host. At this level the process ID (PRID) is ignored, but for simplicity we assume it to be zero as well. Conversely, for programs which run at the level of guest the VMID is taken from SPR register *c.mode*, whereas the PRID consisting of all zeros is reserved. Lastly, for programs at the level of user the VMID and PRID are both taken from the SPR registers, *c.mode* and *c.nmode* resp. Following this convention and using the shorthands

$$\begin{aligned} vmid(c) &= c.mode[31:28] \\ prid(c) &= c.nmode[31:24] \end{aligned}$$

we finally can define the ASID formally:

$$asid(c) = \begin{cases} vmid(c) \circ prid(c) & user(c) \\ vmid(c) \circ 0_8 & guest(c) \\ 0_{12} & host(c). \end{cases}$$

Having introduced the concepts above, one can continue developing new notation and quite naturally arrive at the following partition for the address space of universal page addresses: all addresses with (vm, pr) fields pair “all zeros” fall into the address space of *host*, which we denote by A_H. Addresses with non-zero *vm* field and zero *pr* field fall into the address space of *guest* number $\langle vm \rangle$, which we denote by $A_G(\langle vm \rangle)$. The remaining addresses fall into the address space of *user* number $(\langle vm \rangle, \langle pr \rangle)$, which we denote by $A_U(\langle vm \rangle, \langle pr \rangle)$. Formally we define for $i, j > 0$:

$$\begin{aligned} A_H &= \{upa \in \mathbb{B}^{32} \mid upa.(vm, pr) = 0_{12}\} \\ A_G(i) &= \{upa \in \mathbb{B}^{32} \mid upa.(vm, pr) = i_4 \circ 0_8\} \\ A_U(i,j) &= \{upa \in \mathbb{B}^{32} \mid upa.(vm, pr) = i_4 \circ j_8\}. \end{aligned}$$

Recall, addresses with zero *vm* and non-zero *pr* field we do not consider (see Fig. 10). For convenience we introduce abbreviations to refer to any user/guest address space.

$$A_U(i) \equiv \bigcup_j A_U(i,j)$$
$$A_U \equiv \bigcup_i A_U(i)$$
$$A_G \equiv \bigcup_i A_G(i)$$

Also — in order to save brackets — we directly pass the bit-strings which encode the user/guest numbers (instead of the user/guest numbers) to specify the address space.

3.2 Introduction to NAT

We would like to reuse as much as possible of the previously developed formalism. Fortunately there are very few things that change. Before specifying the mechanism of nested translation formally, we first introduce some basic ideas and machinery in this preliminary section. Starting with changes to the old concepts, we then gradually present what is necessary to form a background for our work.

3.2.1 Ideas behind Nested Translation

Just as the ordinary translation (see Fig. 11) from Sect. 2.4, the nested translation (see Fig. 12) is defined as a process of consecutive walking over the tree-like structure of page tables. These page tables are of the same kind and organized the same way as the prior page tables, but the only and crucial difference is — as we already mentioned — that they are filled with virtual (not physical) addresses. Therefore, for translation of these virtual addresses one needs to incorporate another (ordinary) page table structure, meaning that for the nested translation one needs to use two page table structures simultaneously: a *user page table* for translation of user addresses to the level of guest, and a *guest page table* for (ordinary) translation of guest addresses to the level of host.[2] The origins of these two structures (addresses of their root tables) we keep in special purpose registers $c.npto$ and $c.pto$ resp. for the user and the guest page tables, details of which we already elaborated previously (see Sect. 2.4.1). Note, the address stored in register $c.npto$ is an address from the guest address space, which requires (ordinary) translation like any other virtual address.

When the machine runs at a privilege level higher than the level of host, the special purpose registers involved into the process of address translation should have proper values. At the level of guest this amounts to setting the virtual machine field of register $c.mode$ to the guest's actual VMID, and loading the corresponding page table origin into register $c.pto$. Obviously, at the level of user one additionally needs to set the process field of register $c.nmode$ to the user's actual PRID and load the corresponding page table origin into register $c.npto$. Under this setting the translation mechanisms — which we describe next — can properly serve *universal translation requests*, i.e., translation requests containing the universal page addresses instead of the ordinary ones.

We define this new flavor of translation requests formally by substituting the address $trq.a \in \mathbb{B}^{20}$ of translation request trq by the universal page address $trq.upa \in \mathbb{B}^{32}$. For the new requests we introduce the following (obvious) shorthands:

$$trq.as = trq.upa.as$$
$$trq.pa = trq.upa.pa.$$

For referring to the lower-level components we can use the naming convention of [LOP], e.g.,

[2] Naturally, one needs to maintain a page table structure for every guest, and a page table structure for every user of that guest.

$$trq.vm = trq.as.vm.$$

In a similar fashion we need to update all affected notation. For walks $w \in K_{walk}$ from Sect. 2.4.2 we replace the address $w.a \in \mathbb{B}^{20}$ component by the universal page address $w.upa \in \mathbb{B}^{32}$. Moreover, since in the process of the nested translation the page faults can occur at two different levels (of user and guest), we extend the walks to include an extra "fault" bit, which we attach at the right end. The resulting *universal walks* $w \in K_{uwalk}$ have the following components:

- $w.upa \in \mathbb{B}^{32}$ — universal page address to be translated,
- $w.\ell \in \{100, 010, 001\}$ — number (in one-hot encoding) of walk extensions still required to translate address $w.upa$,
- $w.ba \in \mathbb{B}^{20}$ — base address of the page table to be walked next — if the walk is not faulty and not yet complete — or of the data page — if the walk is not faulty and complete,
- $w.r \in \mathbb{B}^3$ — access rights still remaining, and
- $w.f \in \mathbb{B}^2$ — fault bits indicating that walk extension is no longer possible.

The fault bits are given intuitive names:

$$\begin{aligned} w.f_u &= w.f[1] = w[1] \\ w.f_g &= w.f[0] = w[0]. \end{aligned}$$

Universal walks which have at least one of the latter bits set we call *faulty*.

Definition 1 (Faulty universal walk). *For* $w \in K_{uwalk}$

$$f(w) \equiv w.f_u \vee w.f_g$$

Universal walks which have at most one of the latter bits set are called *well-formed*.

Definition 2 (Well-formed universal walk). *For* $w \in K_{uwalk}$

$$wfu(w) \equiv (w.f_u + w.f_g \leq 1)$$

For the newly obtained walks we abbreviate:

$$\begin{aligned} w.as &= w.upa.as \\ w.pa &= w.upa.pa. \end{aligned}$$

Universal walks with zero *pr* field are called *guest walks* whereas universal walks with non-zero *pr* field are called *user walks*. Guest walks provide translations from the address space of *guest* to the address space of *host*, and user walks translate from the address space of *user* to the address space of *guest*. In what follows we typically represent the guest walks by w_g resp. the user walks by w_u.

Since a number of intermediate results is used in the process of nested translation, we need some more notation. For a pair of universal walks — w_u and w_g below — we define a predicate to express that the walks match, i.e., one of the walks is a (guest) walk of guest j, another one is a (user) walk of user i of guest j, and the base address of user walk equals the virtual address of guest walk.

$$\begin{aligned} match(w_u, w_g) \equiv\ & w_u.ba = w_g.pa \wedge \\ & \exists i > 0 : w_g.upa \in A_G(i) \wedge \\ & \exists j > 0 : w_u.upa \in A_U(i, j) \end{aligned}$$

Of course we call such walks *matching* and, for convenience, we use the following notation to abbreviate the corresponding predicate.

$$w_u \overset{\circ}{=} w_g \equiv match(w_u, w_g)$$

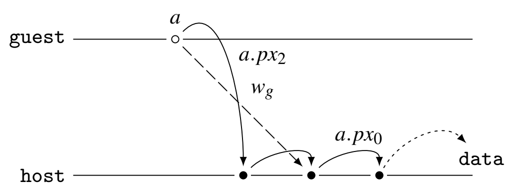


Fig. 11: Process of simple translation

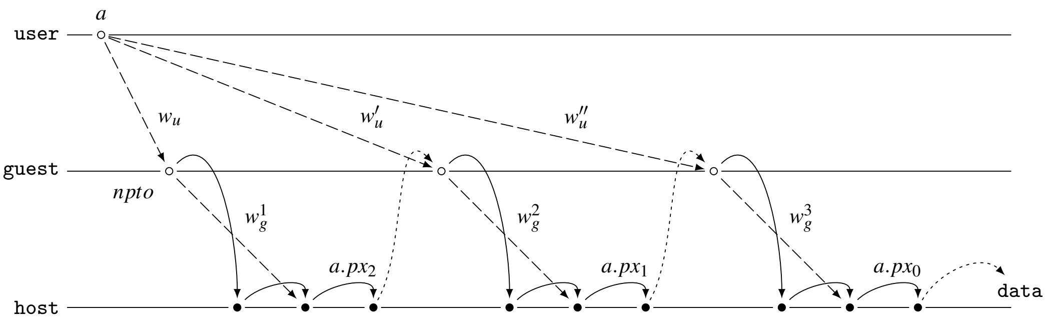


Fig. 12: Process of nested translation

Level of Matching Walks

Recall that walks defined in Sect. 2.4.2 have dedicated components to track the current level, faultiness and rights they provide. We extend these properties in a natural way to pairs of (matching) walks. We start with the *level* of a pair, which we introduce as a ternary number (from zero to eight) formed from the individual levels of walks composing the pair. Formally, for walks w_u and w_g we define

$$\ell(w_u, w_g) = level(w_u) \circ level(w_g).$$

Clearly, the process of nested translation is defined for the pairs of matching walks. Depending on the level of matching walks, we translate — with the ordinary walk extension — through one of the walks composing the pair. At levels 6 and 3 we translate through the user walk (w_u / w'_u) using the completed guest walk (w_g^1 and w_g^2 resp.), otherwise we translate through the guest walk. At level 0 a memory access is performed through the completed user walk (w''_u) using the completed guest walk (w_g^3). Schematically the process of nested address translation is depicted in Fig. 12.

Faultiness of Matching Walks

Though two universal walks are matching, we can use them for the nested translation only if they are not *faulting*. Here we define the faultiness for arbitrary pairs of walks, but later we use it to argue only about matching pairs. Again referring to Fig. 12, we conveniently split the process of nested translation into two phases: translation phase (at levels from 8 to 3) and access phase (at level from 2 to 0). In the translation phase the user walk either is pending to be extended or is being extended (at levels 6 and 3) using the guest walk.[3] In either case the user walk is not yet completed, and we define the pair of walks to be *faulting* if i) some of the walks is faulty or ii) the guest walk does not have rights sufficient to read the user's page table in host memory or to perform the user access.[4] To formalize the second case we introduce a *conditional faultiness* which we define as

[3] Recall, we extend the user walk only after the current guest walk is completed.

[4] The latter situation occurs in practice, for instance, if the user's page table was placed into the memory region where the guest does not have read permissions.

$$f(w_g \mid w_u) \equiv \begin{cases} w_g.r \not\geq 010 & \overline{complete(w_u)} \quad \text{[PT access]} \\ w_g.r \not\geq w_u.r & complete(w_u) \quad \text{[user access].} \end{cases}$$

In the translation phase — when the user walk is not yet completed — the guest walks are used to read the user's page tables, i.e., the read rights suffice. In the access phase either the current guest walk is being extended (at levels 2 and 1) or the final memory access is performed. Anyway, what we care about in this phase are the rights for the memory access which is *not* a read of the user's page table anymore, but an actual memory access from the user's program.

Using the conditional faultiness we formalize the faultiness of a pair (w_u, w_g) of walks as

$$f(w_u, w_g) \equiv f(w_u) \vee f(w_g) \vee f(w_g \mid w_u).$$

With a simple software condition on the content of guests' page tables (via granting the full rights), one could allow guests to manage their address spaces on their own. That would simplify programming on the level of guests (by dropping the limitations on the memory usage) and our definitions as well (by setting $w_g.r$ above to all ones).

3.2.2 Composition of Walks

Just like in case of ordinary translation, to access the virtual memory location at the level of user one has to perform the nested translation of the (virtual) user address. This nested translation results in a pair of matching complete walks — one user walk and one guest walk — which together provide translation for the given user address. After these walks are placed into the specification TLB, they can be used by the stepping function, e.g., in the processor core steps. Note that the specification TLB now stores walks of both types: the user walks together with the guest walks. Formally we describe this in Sect. 3.3, where we give the complete specification of our new machine.

Intuitively, aiming at the hardware acceleration of the process sketched above, one would attempt to reduce the number of walks involved, which basically defines the number of TLB look-ups necessary to obtain the physical address. A possible solution would be to store in the hardware TLB, apart of the ordinary guest walks, the special walks which would translate the user addresses directly (!) to the physical addresses. Though it is simple to implement, this solution requires us to introduce some more notation, so that we can formally argue about this new type of walks.

We start with the notion of *composition* of two walks. We denote this operation by overloading the symbol "$\circ$" and formally define it for walks w_u and w_g as follows:

$$w_u \circ w_g = \begin{cases} w & w_u \overset{\circ}{=} w_g \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} w.upa &= w_u.upa \\ w.\ell &= w_u.\ell \\ w.ba &= w_g.ba \\ w.r &= w_u.r \\ w.f_u &= f(w_u) \\ w.f_g &= f(w_g) \vee f(w_g \mid w_u). \end{aligned}$$

In essence, we transform the user walk s.t. its base address (a virtual address) is remapped to the base address of the guest walk (a physical address). Note that the right bits $w.r$ and the faulty bit $w.f$ are chosen according to resp. the accumulated rights and the faultiness, both defined above in Sect. 2.4.2. Motivated by the latter usage, we call walks obtained via this operation *nested*, and typically represent them by w_n.

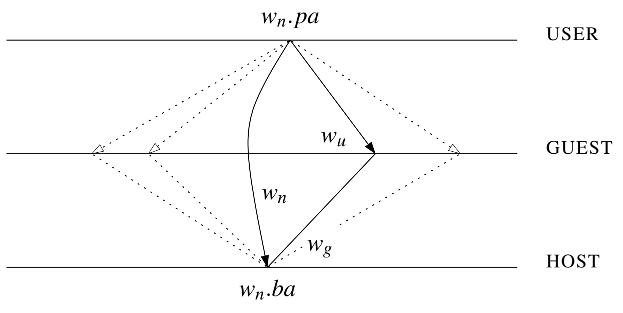


Fig. 13: Possible ways to compose walk w_n

In the obvious way we generalize the new operation for sets $\mathcal{W}_1$ and $\mathcal{W}_2$ of walks:

$$\mathcal{W}_1 \circ \mathcal{W}_2 = \{w_u \circ w_g \mid (w_u, w_g) \in \mathcal{W}_1 \times \mathcal{W}_2 \wedge w_u \stackrel{\circ}{=} w_g\}.$$

Directly from the definitions we derive the following trivial lemmas about the composition of sets of walks.

Lemma 11. *The composition of set $\mathcal{W}_1$ (of walks) with set $\mathcal{W}_2$ (of walks) is exactly the composition of user walks from set $\mathcal{W}_1$ with guest walks from set $\mathcal{W}_2$:*

$$\mathcal{W}_1 \circ \mathcal{W}_2 = \{w \in \mathcal{W}_1 \mid w.upa \in A_U\} \circ \{w \in \mathcal{W}_2 \mid w.upa \in A_G\}$$

Lemma 12. *The composition of union of sets $\mathcal{W}_1$ and $\mathcal{W}_2$ with set $\mathcal{W}_3$ is exactly the union of compositions of $\mathcal{W}_1$ with $\mathcal{W}_3$ and $\mathcal{W}_2$ with $\mathcal{W}_3$:*

$$(\mathcal{W}_1 \cup \mathcal{W}_2) \circ \mathcal{W}_3 \; = \; \mathcal{W}_1 \circ \mathcal{W}_3 \; \cup \; \mathcal{W}_2 \circ \mathcal{W}_3$$

Below we reserve a concise shorthand referring to the generalization above. In case the sets being composed are equal — we consider a single set $\mathcal{W}$ of walks — we abbreviate:

$$\mathcal{W}^{\circ} \; \equiv \; \mathcal{W} \circ \mathcal{W}.$$

The following predicates help us to formalize conditions under which walk extension steps are possible. The first predicate indicates whether walk w is *valid* for the given set of walks $\mathcal{W}$: incomplete, not faulty, and belongs to $\mathcal{W}$ (presumably TLB).

$$valid(\mathcal{W}, w) \; \equiv \; w \in \mathcal{W} \wedge /complete(w) \wedge /f(w)$$

The second predicate indicates whether pair of walks (w_u, w_g) is *valid* for the given set of walks $\mathcal{W}$: matching, not faulty, and both walks from the pair are valid.

$$\begin{aligned} valid(\mathcal{W}, w_u, w_g) \equiv {} & valid(\mathcal{W}, w_u) \wedge w_u \stackrel{\circ}{=} w_g \\ & valid(\mathcal{W}, w_g) \wedge /f(w_u, w_g) \end{aligned}$$

3.2.3 Decomposition of Nested Walks

For the forthcoming section on semantics we need to have sort of an inverse operation to the walk composition: given the nested walk (w_n) decompose it into two matching walks, namely the user walk (w_u) and the guest walk (w_g). This operation we naturally call *walk decomposition*. We define it for the given set $\mathcal{W}$ of walks (containing both user and guest walks) which could be used for the decomposition (note that the resulting guest walk is complete by definition of the composition operation):

$$\begin{aligned} u(w_n, \mathcal{W}) &= \varepsilon\{w \in \mathcal{W} \mid \exists w_g : w \circ w_g = w_n\} \\ g(w_n, \mathcal{W}) &= \varepsilon\{w \in \mathcal{W} \mid \exists w_u : w_u \circ w = w_n\}. \end{aligned}$$

Walk decomposition is inherently not unique, since the same nested walk could be composed out of many different pairs of user and guest walks, as depicted in Fig. 13. So clearly we face the problem, namely that walks w_u and w_g into which walk w_n was decomposed are not well defined. Note, the definition above allows walks which cannot be composed back into w_n (e.g., not matching walks from different pairs).

If one analyzes the situation above carefully, one would notice that it only occurs in practice — set $\mathcal{W}$ contains multiple pairs providing translation from $w_n.pa$ to $w_n.ba$ — in case there are multiple distinct walks which translate from the address space of the same guest to unique (physical) address $w_n.ba$. Which in turn means that multiple distinct locations in virtual memory of the same guest are represented by a single location in the physical memory. In particular, if multiple guest addresses are mapped to the same physical address, semantics of the virtual memory becomes invalid for the level of guests.[5] Nevertheless, programming in such model is reasonable [Hil05], but has to be carried out with caution.

Of course, the problem above can be avoided with certain (software) restrictions on the content of the page tables, but we build our hardware to work for the general case. This forces us to store additional data in the hardware TLB in order to preserve efficiency. These are data about the origin of the nested walks: page addresses of the guest walks used to compose the nested walks.

3.2.4 Overloading Notation

We finish this section with a small list of changes to old notation that we need next. Due to a change of the walk format, clearly we need to update those predicates which operate on walks. The translation requests we already updated in Sect. 3.2.1. Now, for the universal translation requests trq and walks w we redefine the $match$ predicate:

$$match(trq,w) \;\equiv\; (trq.upa = w.upa) \wedge (complete(w) \vee f(w)).$$

Other predicates which we use in the next section ($pfault$ and $gfault$) depend on $match$ internally, so we assume they get updated automatically. In a similar fashion we update the definition of translated memory address:

$$tma(a,w) = w.ba \circ a.po.$$

Note, the latter definition is used only for complete non-faulty universal walks (w) which translate the given universal address (a).

$$(a.upa = w.upa) \wedge complete(w) \wedge /f(w)$$

At this point our formalism is developed well enough to specify the new semantics. This semantics — covering execution in the new translation mode — turns out to be suspiciously similar to the original one [Sch13a]. But that is not a coincidence: in order to achieve the similarity of formulations we invested a lot into notation which allows us to hide the unnecessary complexity inside the formalism. Choosing a simpler but bulkier notation instead would make the text heavier and blur out the intuition, which we want to preserve most. Apart of that, in our opinion, the translation scheme we are adding is rather lengthy than complex.

3.3 MIPS ISA with NAT

Technically speaking, the machine with nested address translation is (in general) very similar to the machine with ordinary address translation developed in Chap. 2. There is no need to introduce new configuration components. For already existing components it is sufficient to adjust their input alphabets and semantics of their steps. In order to save space and — what is more important — to make definitions concise, in this section we discuss only changes to the original specification from Sect. 2.4.

[5] The same problem with the virtual memory semantics occurs for the level of users, though it does not influence the uniqueness of the decomposition.

3.3.1 TLB Component

In the previous section — in the course of defining the process of nested translation — at some point we had to switch to the new format for representation of walks (see Sect. 3.2.1). In the new format we supplied the virtual address with a special field (ASID), which identifies the address space to which that particular virtual address belongs,[6] and an extra fault bit, to distinguish between faults occurring at the levels of user and guest. Clearly, the TLB component has to be changed appropriately to store walks of the universal format.

Recall, all walks of the universal format are collected into set K_{uwalk}. Configurations of the *universal TLB* are obviously then taken from the powerset of K_{uwalk}.

$$c.tlb \in K_{utlb} = 2^{K_{uwalk}}$$

Thus, the universal TLB component now stores walks of both types: guest walks and user walks. For convenience we partition our new TLB according to the type of walks stored:

$$\begin{aligned} tlb_G(c) &= \{w \in c.tlb \mid w.upa \in A_G\} \\ tlb_U(c) &= \{w \in c.tlb \mid w.upa \in A_U\}. \end{aligned}$$

As in the system mode before, at the level of host the TLB component does not perform any steps. At the level of guest the TLB component is allowed to perform essentially the same steps as in the former user mode: creation and extension of the guest walks. Finally, at the level of user the TLB component is allowed to perform all kind of steps. As a result, very similar steps are now allowed at both levels — of guest and user — and to distinguish between those steps we change the TLB portion Σ_{tlb} of the machine's input alphabet Σ.

For walk creation steps we extend the input with one more 'bit' to distinguish initialization of the guest walks from initialization of the user walks:

$$\{winit\} \times \mathbb{B}^{20} \times \{guest, user\} \subset \Sigma_{tlb}.$$

Intuitively, semantics of these steps stays unchanged: an initialized walk for the given virtual address is included into the TLB. But now we change the initialization part: depending on the new parameter passed — a guest or a user walk is created — either the guest's page table origin is chosen to initialize the walk or the user's one. Moreover, the virtual address is prepended with its address space identifier; the walk initialization function has to be overloaded appropriately to handle universal page addresses. Below we specify the effect of this transition formally, assuming that x there denotes the machine's input.

$$c'.tlb = c.tlb \cup \begin{cases} \{initw(vmid(c) \circ 0_8 \circ a, pto(c))\} & x = (winit, a, guest) \\ \{initw(vmid(c) \circ prid(c) \circ a, npto(c))\} & x = (winit, a, user) \end{cases}$$

For walk extension steps we change the input to contain either one or two walks:

$$\{wext\} \times K_{uwalk} \times (\{\perp\} \cup K_{uwalk}) \subset \Sigma_{tlb}.$$

Assume again that x is the machine's input.

$$x = (wext, w_1, w_2) \in \Sigma_{tlb}$$

The first walk passed (w_1) is always the one to be extended. In case it is a guest walk, an ordinary walk extension step is to be performed. In this case the second walk (w_2) necessarily has to be undefined.

$$w_1.upa \in A_G \rightarrow w_2 = \perp$$

If w_1 is a user walk, again an ordinary walk extension step is to be performed, but this time walk w_1 is first composed with walk w_2. In this case the second walk has to be (at least) defined.

[6] Recall, due to the virtual memory, each user or guest program operates within its own isolated address space.

$$w_1.upa \in A_U \;\rightarrow\; w_2 \neq \bot$$

Note that both walks are expected to be in the universal walk format. Therefore the walk extension function has to be overloaded as well to fit the new arguments' format. The semantics of walk extension steps is given below.

$$c'.tlb \;=\; c.tlb \cup \begin{cases} \{wext(w_1, c.m)\} & x = (wext, w_1, \bot) \\ \{wext(w_1 \circ w_2, c.m)\} & x = (wext, w_1, w_2) \wedge w_2 \neq \bot \end{cases}$$

The first line of the definition above simply repeats the original behavior, whereas the second line formalizes the result of the nested walk extension. In the latter case we make use of our new notation and pass to the overloaded walk extension function the composition of two walks. Obviously, we require those walks to be matching. For intuition we refer the reader to the introductory Sect. 3.2.2, where we describe the process of nested translation in more detail.

Below we collect in one place all guard conditions on the TLB steps, including those we introduced in Sect. 2.4.4 and those we informally imposed right above. To every condition below we attach a short description.

- Steps of the TLB component are allowed at the levels of guest and user. At the level of guest only two TLB steps are allowed: initialization of the guest walk and extension of the guest walk. At the level of user any TLB steps are allowed.

$$x \in \Sigma_{tlb} \rightarrow guest(c) \vee user(c)$$
$$x \in \Sigma_{tlb} \rightarrow x \in \begin{cases} \{(winit, a, guest), (wext, w_1, \bot)\} & guest(c) \\ \{(winit, a, *), (wext, w_1, w_2)\} & user(c) \end{cases}$$

- Only incomplete and non-faulty walks from the TLB are allowed to be extended (recall that we always extend the first walk). That is the first condition below, which was inherited and without changes fits the new case of nested translation. At the level of user we allow extension of walks of the current user and guest; at the level of guest we allow extension of walks of the current guest only. That is the second condition, which is caused by introduction of the ASID fields. In order to formulate it in a more intuitive way, we use our new notation from Sect. 3.2 and the following shorthands to specify the current ASID:

$$asid(c) = i_4 \circ j_8.$$

Note, if on the steps of walk initialization we allowed to specify the ASID fields, we would have to add a similar guard condition restricting those steps.

$$x = (wext, w_1, w_2) \rightarrow valid(c.tlb, w_1)$$
$$x = (wext, w_1, w_2) \rightarrow w.upa \in \begin{cases} A_G(i) & guest(c) \\ A_G(i) \cup A_U(i,j) & user(c) \end{cases}$$

- On the nested translation steps — extensions of the user walks — the walk to be extended and the walk used for extension are matching. The latter one is a complete walk from the TLB which does not lead to a faulting composition (see Sect. 3.2.1).

$$x = (wext, w_1, w_2) \rightarrow valid(c.tlb, w_1, w_2)$$

These guard conditions apply simultaneously, i.e., all of the conditions above must be satisfied for a TLB step to be possible. Later on, whenever we need to argue about TLB steps within a lengthy proof, we would like to have these conditions in a more concise form. For that reason we develop some more notation. Components of the machine's input receive the following names:

$$x = \begin{cases} x.(t, a, l) & x \in \Sigma_{winit} \\ x.(t, w_1, w_2) & x \in \Sigma_{wext} \end{cases}$$

where

- $.t$ gives the type of the performed step; Obviously we have

$$x.t \in \{winit, wext\}.$$

- $.a$ gives the page table origin used for creation of a new walk; Clearly

$$x.a \in \mathbb{B}^{20}.$$

- $.l$ gives the level for which a new walk is created; We clearly have

$$x.l \in \{guest, user\}.$$

- $.w_1$ gives the universal walk extended in the corresponding step

$$x.w_1 \in K_{uwalk},$$

- while $.w_2$ gives the universal walk used for extension of walk w_1 (if not $\perp$):

$$x.w_2 \in \{\perp\} \cup K_{uwalk}.$$

Using the notation above we can equivalently reformulate the guard conditions for TLB steps as follows. In order to separate conditions that restrict guest walks from conditions that restrict user walks, we introduce two dedicated predicates. Conditions restricting guest walks are covered by

$$T_G(c,x) \equiv /host(c) \wedge \begin{cases} x.l = guest & x.t = winit \\ x.w_1.vm = vmid(c) \wedge valid(c.tlb, x.w_1) \wedge x.w_2 = \perp & x.t = wext. \end{cases}$$

The remaining conditions, restricting the user walks, are covered by

$$T_U(c,x) \equiv user(c) \wedge \begin{cases} x.l = user & x.t = winit \\ x.w_1.as = asid(c) \wedge valid(c.tlb, x.w_1, x.w_2) & x.t = wext. \end{cases}$$

As a result, the guard conditions for TLB steps are now satisfied iff they are satisfied either for guest or for user walks.

$$T(c,x) = T_G(c,x) \vee T_U(c,x)$$

Taking these guard conditions into account, walk initialization and walk extension functions guarantee that for the guest and user walks stored in the TLB we resp. have

$$w \in tlb_G(c) \;\rightarrow\; w.f_u = 0$$
$$w \in tlb_U(c) \;\rightarrow\; w.f_g = 0.$$

The latter obviously implies that all walks stored in the TLB are well-formed.

Invariant 1.

$$w \in c.tlb \;\rightarrow\; wfu(w)$$

3.3.2 Translation Accesses

In the process of address translation, the memory-hosted structures called page tables are traversed as described in Sect. 2.4. Thus, on steps of the walk extension, i.e., when the specification machine receives input

$$x = (wext, w_1, w_2) \in \Sigma_{tlb},$$

the memory is accessed at addresses

$$ptea(x) = \begin{cases} ptea(w_1 \circ w_2) & w_2 \neq \perp \\ ptea(w_1) & \text{otherwise.} \end{cases}$$

For the page table entries used on steps of the walk extension, according to Sect. 3.3.1, we abbreviate

$$pte(c,x) = \begin{cases} pte(w_1 \circ w_2, c.m) & w_2 \neq \perp \\ pte(w_1, c.m) & \text{otherwise} \end{cases}$$

and naturally obtain the following.

Lemma 13.

$$pte(c,x) = c.m_4(ptea(x))$$

Proof of lemma 13.

$$\begin{aligned}
pte(c,x) &= \begin{cases} pte(w_1 \circ w_2, c.m) & w_2 \neq \bot \\ pte(w_1, c.m) & \text{otherwise} \end{cases} \\
&= \begin{cases} c.m_4(ptea(w_1 \circ w_2)) & w_2 \neq \bot \\ c.m_4(ptea(w_1)) & \text{otherwise} \end{cases} \\
&= c.m_4(ptea(x))
\end{aligned}$$

□

Analogous to Sect. 2.2.3, we rephrase the read of page table entry $pte(c,x)$ in terms of translation access $tacc(x)$ to the line addressable version $\ell(c.m)$ of the memory.

$$\begin{aligned}
tacc(x).a &= ptea(x).l \\
tacc(x).r &= 1
\end{aligned}$$

As the output to the translation access we have

$$tmout(c,x) = dataout(\ell(c.m), tacc(x)).$$

Using properties of the memory embedding, for the output to the translation access we derive the following.

Lemma 14.

$$pte(c,x) = \begin{cases} tmout(c,x)_H & ptea(x)[2] \\ tmout(c,x)_L & \textit{otherwise} \end{cases}$$

Proof of lemma 14.

$$\begin{aligned}
pte(c,x) &= c.m_4(ptea(x)) \qquad \text{(lemma 13)} \\
&= c.m_4(ptea(x).l \circ ptea(x)[2] \circ 00) \qquad \text{(definition)} \\
&= \begin{cases} c.m_4(ptea(x).l \circ 100) & ptea(x)[2] \\ c.m_4(ptea(x).l \circ 000) & \text{otherwise} \end{cases} \\
&= \begin{cases} c.m_8(ptea(x).l \circ 0_3)_H & ptea(x)[2] \\ c.m_8(ptea(x).l \circ 0_3)_L & \text{otherwise} \end{cases} \qquad \text{(definition)} \\
&= \begin{cases} \ell(c.m)(ptea(x).l)_H & ptea(x)[2] \\ \ell(c.m)(ptea(x).l)_L & \text{otherwise} \end{cases} \qquad \text{(definition)} \\
&= \begin{cases} dataout(\ell(c.m), tacc(x))_H & ptea(x)[2] \\ dataout(\ell(c.m), tacc(x))_L & \text{otherwise} \end{cases} \qquad \text{(definition)} \\
&= \begin{cases} tmout(c,x)_H & ptea(x)[2] \\ tmout(c,x)_L & \text{otherwise} \end{cases} \qquad \text{(definition)}
\end{aligned}$$

□

3.3.3 Faults of NAT

In the previous section we focused our attention on cases in which steps of the nested translation are actually performed, i.e., the walk extension function is involved to access the memory — to read the user's page table. Unfortunately, that is not the case in general. For instance, those guest's pages which contain the user's page tables can be not-accessible due to insufficient rights or simply because they were temporarily swapped-out[7]. Therefore, the situations

[7] Clearly, such situations are possible unless the hypervisor allocates for guests (operating systems) sufficient number of *memory-locked* pages, or provides an alternative mechanism that allows guests to specify the memory regions with user page table structures as non-swappable. For details we refer to [Vir18].

in which a step of the nested translation cannot be performed may occur even though all user's pages are present in the memory. Such situations we call *faults* of the nested translation or *nested faults* for short.

Another possible interpretation of the nested faults — especially taking into account their effect — is to consider them simply as yet another source of ordinary page faults which affects only the nested translation. A page fault is triggered at the level of user if a faulty walk (w_Y) is passed on the processor core step. In case the latter walk is guest-faulty

$$w_Y.f_g = 1,$$

the interrupt is intercepted and the machine switches to the level of host (see Sect. 3.1.2). Thus, it suffices to have the same number of interrupt causes as before in order to handle the page faults which occur due to the nested translation. In Sect. 3.3.5 we cover the details of instruction execution in presence of nested faults. In the remainder of this section we present the semantics of the processor core steps, starting with the changes to the input alphabet (Sect. 3.3.4).

3.3.4 Processor Core

Just as before, on the processor core step either

i) a current instruction is executed "uninterrupted", or
ii) a jump to the interrupt service routine is performed due to an interrupt.

In both cases the occurrence of an interrupt entirely depends on the current machine's input and the processor local configuration, which is so far the configuration of the processor core together with the TLB.[8] On the level of semantics this means that either one transition function is applied or the other. This behavior should not change after extension of the ISA with a new translation mode, and it does not. Also we want to preserve the old format of the machine's input for the processor core steps. So, we leave the input alphabet Σ_{core} unchanged, but as before we switch to the universal walk format:

$$\Sigma_{core} = (\{\bot\} \cup K_{uwalk})^2 \times \mathbb{B}^2.$$

Obviously this works for the processor core steps made at the level of host or guest. But what about the level of user, where the machine is supposed to perform the nested translation...? The trick lies in the following: in the latter case (level of user) we pass the nested walks, which we formally specified in Sect. 3.2.1. Recall that nested walks are obtained using the operation of walk composition we introduced above, where the user walks were "prolonged" via the matching guest walks. Formally, these nested walks are not present in the specification TLB and, strictly speaking, they are a "virtual" concept for the "real" specification configurations[9]. However, using the notation we introduced together with the nested walks, we can form the exact set of all possible nested walks which could be obtained from the current TLB. In the new notation this set is $c.tlb^{\circ}$. As simple as that.

Using notation of [LOP] from Sect. 2.4.5, we denote the machine's input by

$$x = (w_I, w_E, eev) \in \Sigma_{core}$$

and expect it to contain guest walks from the TLB on the level of guest

$$guest(c) \land w_Y \neq \bot \;\rightarrow\; w_Y \in tlb_G(c)$$

and nested walks from the composed TLB on the level of user.

$$user(c) \land w_Y \neq \bot \;\rightarrow\; w_Y \in c.tlb^{\circ}$$

[8] The configuration of the memory does not influence occurrence of the interrupts (see Sect. 2.3).

[9] For the hardware configurations they appear to be the format of data stored in the hardware TLB cache. We discuss this in more detail in the forthcoming chapter on hardware (Chap. 4), but already now it might be clear that usage of the same format for ISA and hardware simplifies the future correctness proof considerably.

Table 11: Meaningful ISA signals (denoted by ✓) for various interrupt levels

Input/signal		Interrupt level											
		reset	*e*	*malf*	*pff*	*gff*	*ill*	*sys*	*ovf*	*malm*	*pfm*	*gfm*	∞
eev	*exec fetch*	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	...	▨	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
w_I	...	▨	▨	▨	✓	✓	✓	✓	✓	✓	✓	✓	✓
	pma_I *I*	▨	▨	▨	▨	▨	✓	✓	✓	✓	✓	✓	✓
w_E	...	▨	▨	▨	▨	▨	▨	▨	▨	▨	✓	✓	✓
	pma_E *lres*	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	✓

Note, above and in the sequel, we follow the convention from [LOP] for index Y, which ranges over $\{I,E\}$. In addition, we introduce a similar convention for index X, which we use to iterate over $\{G,U\}$. We specify the guards formally in Sect. 3.3.5. Again, below we list only those things that change compared to the original specification in Sect. 2.4.5, and again, we assume that x denotes the machine's current input.

In the presence of interrupts several extra arguments are necessary to specify the instruction execution formally. Since according to specification from Sect. 2.3.3 on the non-continue type interrupts the execution of instruction is immediately terminated, signals that would normally be utilized by the instruction are not used. For instance, we define the following two auxiliary signals to be always used.

$$exec(c,x) \equiv jisr(c,x) \to cont(c,x)$$
$$fetch(c,x) \equiv jisr(c,x) \to il(c,x) > 4$$

In Table 11 we visualize this by listing the latter signals in the topmost row. Note, this row is left entirely white, meaning that signals assigned to this row are always meaningful, independent of the interrupt level. In the rows below, which are partially grayed out, we collect the remaining signals. We distribute them according to the interrupt levels at which they become meaningful. Since it does not make sense to make formal arguments about things which are not meaningful, like signals that are not used, in the proofs we simply ignore such unused signals.

By analyzing the specification from Chap. 2, Table 11 can be easily populated and made precise. Though not all signals are listed, the number of rows in the table suggests that for the interrupts considered all signals fall into one out of six possible groups. To show the instructions are executed correctly one has to argue about signals from all groups (rows of the table) that are meaningful at the corresponding interrupt levels and, moreover, utilized by the executed instructions. Indeed, for every ISA signal we specify conditions under which it is *used*. For instance, for the walks passed as the machine's input (w_I and w_E) we specify the used predicates to be as follows.

$$used(w_I,c,x) \equiv /host(c) \wedge il(c,x) > 2$$
$$used(w_E,c,x) \equiv /host(c) \wedge il(c,x) > 8 \wedge mop(c,x)$$

Thus, the first walk (w_I) is used whenever the machine runs in translated mode and the sampled interrupts are of interrupt level 3 or higher. Similarly, the second walk (w_E) is used whenever the machine run in translated mode, the sampled interrupts are of interrupt level 9 or higher. Note, in absence of interrupts the interrupt level by definition is equal to $+\infty$ (see Sect. 2.3.3). The used predicates for most of the signals are given in [LOP] in the form of tables.

To finish definition of semantics for the processor core steps we proceed as follows. In the next section we discuss changes of semantics for the "uninterrupted" execution of instructions. To ease the presentation, details covering execution of the *invlpg* and *flusht* instructions are taken out of the scope and considered later, in Sect. 3.3.6. In the latter section we also update definition of the illegal interrupt. In short, the new definition allows guests to execute privileged instructions under certain restrictions. Details of the interrupt handling are covered in Sect. 3.3.7.

3.3.5 Execution of Instructions

We start with the translation requests and modify them in an obvious way to work with the universal page addresses. Formally we redefine them as follows:

$$trq_I(c,x) = (asid(c) \circ ia(c,x).pa, 110)$$
$$trq_E(c,x) = (asid(c) \circ ea(c,x).pa, 01\texttt{s})$$

where the write-bit s of the requested rights is set only if the store or CAS access takes place. Recall that $asid(c)$ denotes the ASID of the current ISA configuration.

$$\texttt{s} \equiv s(c,x) \vee cas(c,x)$$

Next we cover the generation of interrupt signals, and essentially, there are no changes to the generation mechanism developed in Sect. 2.3. As before, interrupts are discovered gradually — under the priorities specified in Table 9 — and once an interrupt is discovered, the execution of the current instruction might be terminated if the resume type of the causing interrupt is not *continue*. Formally, we change the definition of page faults to reflect the obvious changes to the machine's mode:

$$pff(c,x) \equiv used(w_I,c,x) \wedge pfault(trq_I(c,x),w_I)$$
$$pfm(c,x) \equiv used(w_E,c,x) \wedge pfault(trq_E(c,x),w_E).$$

Note, in the above definition the page faults are signaled in case the walks passed are faulty. Otherwise, in case the rights provided by these walks are insufficient to perform an access, the *general-protection faults* are signaled. New faults are defined as follows:

$$gff(c,x) \equiv used(w_I,c,x) \wedge gfault(trq_I(c,x),w_I)$$
$$gfm(c,x) \equiv used(w_E,c,x) \wedge gfault(trq_E(c,x),w_E).$$

Since the main idea of NAT is to make the page faults that occur due to translations of intermediate guest addresses completely "invisible" at the level of guest, these page faults are intercepted by the mechanism we developed previously, in Sect. 3.1.2.

$$icpt(c,x) \equiv user(c) \wedge \bigvee_Y used(w_Y,c,x) \wedge w_Y.f_g$$

In the absence of interrupts, the semantics of instruction execution at the level of user — when the machine runs in the new translation mode — essentially stays unchanged. For all instructions — except of *invlpg*s and *flush*es — the only thing we update is the definition of physical memory addresses used for the instruction fetch and memory access:

$$pma_I(c,x) = \begin{cases} ia(c,x) & host(c) \\ tma(asid(c) \circ ia(c,x), w_I) & \text{otherwise} \end{cases}$$
$$pma_E(c,x) = \begin{cases} ea(c,x) & host(c) \\ tma(asid(c) \circ ea(c,x), w_E) & \text{otherwise.} \end{cases}$$

In the end we collect in one place all of the guard conditions on the inputs of processor core steps. We conveniently split these guards into two categories: for the instruction fetch and for the memory access. For every guard condition below we give a short description.

- At the levels of user or guest — with the address translation — in the absence of interrupts of interrupt level 2 or lower, the first walk passed to the machine (via $x \in \Sigma_{core}$) is the one matching the translation request for the current instruction address: at the level of guest it is a matching walk from the regular TLB, whereas at the level of user — a matching walk from the composed TLB.

$$/host(c) \wedge il(c,x) > 2 \rightarrow match(trq_I(c,x),w_I) \wedge w_I \in \begin{cases} tlb_G(c) & guest(c) \\ c.tlb^{\circ} & user(c) \end{cases}$$

- The guard conditions on the second walk passed are analogous to those above, but apply only on memory operations ($mop(c,x)$). In contrast to the conditions above, here we require the absence of interrupts of interrupt level 8 or lower.

$$/host(c) \wedge il(c,x) > 8 \;\rightarrow\; match(trq_E(c,x), w_E) \wedge w_E \in \begin{cases} tlb_G(c) & guest(c) \\ c.tlb^{\circ} & user(c) \end{cases}$$

Similarly to Sect. 3.3.1, we reformulate the guard conditions for processor core steps in a more concise way. This time the definitions become even shorter: conditions restricting walk w_Y passed as the machine's input are covered by predicate

$$\Phi_Y(c,x) \equiv used(w_Y, c, x) \;\rightarrow\; match(trq_Y(c,x), w_Y) \wedge w_Y \in \begin{cases} c.tlb^{\circ} & user(c) \\ c.tlb & guest(c). \end{cases}$$

Clearly, the guard conditions for the processor core steps are now satisfied iff they are satisfied for both walks contained in the machine's input.

$$\Phi(c,x) = \Phi_I(c,x) \wedge \Phi_E(c,x)$$

3.3.6 Overloading Instructions

In the previous chapters on semantics with interrupts, we had to distinguish only between two possibilities, namely if the machine runs in translated mode or not. Depending on that, certain instructions could be executed only at the privileged level (*system mode*), causing interrupts (illegal instruction) at the level of user programs (*user mode*). Now, having added a new level of privilege ("in the middle"), we need to change the old definitions to suit our new setting. Note, this won't be just a syntactical patch caused by the change of naming; we want to allow at the level of guest usage of the instructions forbidden at the level of user. However, the effect of those instructions can *not* be the same as when they are executed at the level of host, which causes us to overload them for guests.

Clearly, we argue only about instructions for operating system support. In Sect. 2.1.1 such instructions were formally introduced and collected in Table 2. Here, we are interested in the following ones:

- $flusht$ — flushes all translations cached in the TLB, and
- $invlpg$ — flushes the translations for address rt and ASID rs.

In order to define the semantics for the $invlpg$ and $flusht$ instructions, we introduce the following shorthands for the ASID and the page address to be invalidated respectively.

$$\begin{aligned} inva(c,x).as &= \begin{cases} vmid(c) \circ A(c,x)[27:20] & guest(c) \\ A(c,x)[31:20] & \text{otherwise} \end{cases} \\ inva(c,x).pa &= B(c,x)[31:12] \end{aligned}$$

As it might be clear from the last lines, the $invlpg$ instruction uses its first parameter — GPR register rs — to restrict the victim TLB translations to those having their fields pair (vm, pr) equal to the ASID provided by the parameter. Note, at the level of guest the first four bits of the "invalidation" ASID are substituted by the current VMID. This is in order to not let guests invalidate TLB translations of one another. As before, the "invalidation" page address is taken from the GPR register rt.

$$invlpg(c,x) \;\rightarrow\; c'.tlb = \{w \in c.tlb \mid w.as = inva(c,x).as \;\rightarrow\; w.pa \neq inva(c,x).pa \wedge complete(w)\}$$

$$flusht(c,x) \;\rightarrow\; c'.tlb = \begin{cases} \{w \in c.tlb \mid w.vm = vmid(c) \;\rightarrow\; w.pr = 0_8\} & guest(c) \\ \emptyset & \text{otherwise} \end{cases}$$

Note that $flusht$ instruction executed at the level of guest flushes all TLB translations with the *vm* field equal to the current VMID except those with the *pr* field equal to zero. The latter ones translate from the level of guest to the level of host, and therefore must not be manipulated by guests.

3.3.7 Interrupt Mechanism

Overall the interrupts mechanism stays unchanged. We touch only two aspects:

- we plug in the new definition of the illegal interrupt. Thus, at the level of guest, we allow the *move* instructions partially (not with all arguments). Similarly, we allow guests to use the $invlpg$ and $flusht$. Note that the latter instructions were overloaded in Sect. 3.3.6.

$$\begin{aligned} ill(c,x) \equiv\ & undefined(c,x) \vee \\ & user(c) \wedge eret(c,x) \vee \\ & user(c) \wedge (flusht(c,x) \vee invlpg(c,x)) \vee \\ & user(c) \wedge move(c,x) \wedge /(movg2s(c,x) \wedge xad(c,x) \in \{\texttt{cdata}\}) \vee \\ & guest(c) \wedge invlpg(c,x) \wedge inva(c,x) \in A_G \vee \\ & guest(c) \wedge movg2s(c,x) \wedge xad(c,x) \in \{\texttt{pto}, \texttt{mode}, \texttt{nmode}\} \vee \\ & host(c) \wedge movg2s(c,x) \wedge xad(c,x) \in \{\texttt{mode}\} \end{aligned}$$

- we integrate the new definitions spawned by the intercept mechanism. Recall, in Sect. 3.1.2 we specified the effects of *jisr* and *eret* on the machine's state (mode registers).

In all other aspects our "new" mechanism repeats the one described in Sect. 2.3. Note, in Sect. 2.3.3 we already changed the semantics for the exception cause register to store the hardware analogue of the interrupt level ($il(c,x)$).

$$jisr(c,x) \ \rightarrow\ c'.eca = 0_{21} \circ f1(mca(c,x))$$

This allows us to keep the simulation relation for the processor core in its original and usual form. For the same purpose we update semantics for the exception data register. Thus, in case the executed instruction was not fetched, the effective address is clearly not meaningful, and therefore the register is simply filled with zeros.

$$jisr(c,x) \ \rightarrow\ c'.edata = ea(c,x) \wedge fetch(c,x)$$

3.4 Simplified Semantics

In fact, we want to reformulate the specification so that we can argue about the TLB transitions in a more structured manner. There are two kinds of transitions: active and passive. For the TLB component they are represented resp. by addition and drop of the translations[10]. Active transitions of the TLB are triggered explicitly, when the stepping function of the entire machine takes the corresponding values. In turn, passive transitions of the TLB are triggered within the processor core transitions, whenever an invalidating instruction is executed. Note that transitions of both types use certain data from the machine state, i.e., configurations of the processor core and memory (see Sect. 3.3). Aiming at increased modularity of the hardware correctness proofs, we introduce an alternative version of the TLB specification which allows to abstract from the particular machine type. This removes the necessity to repeat the proofs for the MMU component later in this chapter, where we argue about correctness of different machines which contain MMUs for the nested address translation.

[10] This changes if the *access* and *dirty* bits are taken into consideration. In the latter case more active TLB transitions appear.

3.4.1 General Semantics for TLB

We refer to the specification below as the *general semantics* for TLB. In a nutshell it is the same (equivalent) specification as given in Sect. 3.3. We refer to the latter semantics as the *original semantics* for convenience. Configurations of the TLB component (tlb) remain unchanged and still are taken from the same configuration set.

$$tlb \in K_{utlb}$$

The next state configuration is given by the following transition function:

$$\delta_{tlb} : K_{utlb} \times (\Sigma_{add} \cup \Sigma_{drop}) \rightarrow K_{utlb}$$

where the input alphabet is split into

$$\begin{aligned} \Sigma_{add} = {} & \{winit\} \times \mathbb{B}^{32} \times \mathbb{B}^{20} \cup \\ & \{wext\} \times K_{uwalk} \times \mathbb{B}^{32} \end{aligned}$$

and

$$\Sigma_{drop} = \{drop\} \times (\mathbb{B}^{32} \cup \mathbb{B}^{4} \cup \{all\}).$$

Thus, for addition of new walks by initialization we pass the universal page address ($upa \in \mathbb{B}^{32}$) and a page table origin ($pto \in \mathbb{B}^{20}$). For addition of new walks by walk extension we pass the extended walk ($w \in K_{uwalk}$) and a page table entry ($pte \in \mathbb{B}^{32}$). In order to drop the stored walks we either pass a universal page address ($upa \in \mathbb{B}^{32}$), a virtual machine ID ($vm \in \mathbb{B}^{4}$), or a special keyword (*all*) to flush the TLB.

$$\begin{aligned} y \in \Sigma_{add} &\rightarrow y \in \{(winit, upa, pto), (wext, w, pte)\} \\ y \in \Sigma_{drop} &\rightarrow y \in \{(drop, upa), (drop, vm), (drop, all)\} \end{aligned}$$

In this way the next configuration of the TLB component is completely determined by the current TLB configuration and an external input from the latter two sets. Also, transitions in the general semantics are both triggered explicitly by the external inputs, i.e., are both of active type. Recall that active transitions of a component are called steps. For convenience we introduce the following abbreviations for steps in the general semantics which add ($tadd$) and drop ($tdrop$) translations.

$$\begin{aligned} tadd(y) &\equiv y \in \Sigma_{add} \\ tdrop(y) &\equiv y \in \Sigma_{drop} \end{aligned}$$

Finally, we give the specification of the general semantics in the following two lines.

$$\begin{aligned} tadd(y) &\rightarrow tlb' = tlb \cup \{w(y)\} \\ tdrop(y) &\rightarrow tlb' = tlb \setminus \mathcal{I}(y) \end{aligned}$$

Intuitively, sets $\{w(y)\}$ and $\mathcal{I}(y)$ abbreviate the walks which are resp. added to/dropped from the TLB on the corresponding steps. We defined these sets formally in the next section. Only one guard condition remains meaningful in the general semantics. It concerns the steps of walk extension, which are still restricted to all incomplete and non-faulty walks from the TLB.

$$y = (wext, w, pte) \rightarrow valid(tlb, w)$$

3.4.2 Added, Dropped, and Ragged Walks

We start with the walk which is added by the corresponding TLB steps ($tadd$).

Definition 3 (Added Walk).

$$w(y) = \begin{cases} winit(upa, pto) & y = (winit, upa, pto) \\ wext(w, pte) & y = (wext, w, pte) \end{cases}$$

In the latter definition "*winit*" and "*wext*" are those functions which we later turn into the hardware circuits of the MMU component. Their exact specification is as follows. For a universal page address $upa \in \mathbb{B}^{32}$ and a page table origin $pto \in \mathbb{B}^{20}$ we define the output w of the function *winit* as

$$\begin{aligned} winit(upa, pto) &= w.(upa, \ell, ba, r, f_u, f_g) \\ &= (upa, 100, pto, 111, 0, 0). \end{aligned}$$

Recall, universal walks were formally defined in Sect. 3.2.1. Extension of a universal walk $w \in K_{uwalk}$ using a page table entry $pte \in \mathbb{B}^{32}$ is given by the function *wext*, which outputs either a faulty or extended walk, depending on whether the target page is present in the memory or not.

$$\begin{aligned} \overline{pte.p} \;\to\; wext(w, pte) &= \begin{cases} w[f_u := 1] & w.upa \in A_U \\ w[f_g := 1] & w.upa \in A_G \end{cases} \\ pte.p \;\to\; wext(w, pte) &= w'.(upa, \ell, ba, r, f_u, f_g) \\ &= (w.upa, 0 \circ w.\ell[2:1], pte.ba, w.r \wedge pte.r, 0, 0). \end{aligned}$$

Note, in case the target page is not present ($/pte.p$), the level of the extended walk becomes the level of the walk extension. A trivial lemma follows.

Lemma 15.

$$wext(w, m) \;=\; wext(w, pte(w, m))$$

Next we define the sets of walks which are invalidated on the corresponding TLB steps ($tdrop$).

Definition 4 (Dropped Walks).

$$\mathcal{I}(y) = \begin{cases} \mathcal{I}_{\mathcal{V}}(y) \cup \mathcal{I}_{\mathcal{C}}(y) & y = (drop, upa) \\ \mathcal{I}_{\mathcal{D}}(y) & y = (drop, vm) \\ K_{utlb} & y = (drop, all) \end{cases}$$

where

$$\begin{aligned} \mathcal{I}_{\mathcal{V}}(y) &= \{w \mid y = (drop, upa) \;\wedge\; w.upa = upa\} \\ \mathcal{I}_{\mathcal{C}}(y) &= \{w \mid y = (drop, upa) \;\wedge\; w.as = upa.as \wedge \overline{w.\ell[0]}\} \\ \mathcal{I}_{\mathcal{D}}(y) &= \{w \mid y = (drop, vm) \;\;\wedge\; w.upa \in A_U(vm)\} \end{aligned}$$

Into set $\mathcal{I}_{\mathcal{V}}$ we collect all walks which translate the invalidated address (*upa*), into set $\mathcal{I}_{\mathcal{C}}$ — all walks incomplete walks which translate addresses from the invalidated address space (*upa.as*). The definitions of the latter two sets are only meaningful in case an individual address is invalidated. If an entire virtual machine is invalidated, we collect all user walks of the invalidated machine (*vm*) into set $\mathcal{I}_{\mathcal{D}}$.

For reasons that become clear later we formalize the following set of walks: all user walks from the TLB which are composable with the invalidated or incomplete walks.

Definition 5 (Ragged Walks).

$$\mathcal{R}(y) \;=\; \{w \mid \{w\} \circ (\mathcal{I}_{\mathcal{V}}(y) \;\cup\; \mathcal{I}_{\mathcal{C}}(y)) \neq \emptyset\}$$

We call walks from the latter set *ragged*, in a sense that these walks do not count in construction of the composed TLB once the sets of invalidated and incomplete walks are dropped. Obviously, the interesting case is when a guest address is invalidated, since otherwise the set of ragged walks is empty by definition.

$$y = (drop, upa) \wedge upa \in A_U \;\to\; \mathcal{R}(y) = \emptyset$$

We infer the following lemma about the sets of invalidated ($\mathcal{I}_{\mathcal{V}}$), incomplete ($\mathcal{I}_{\mathcal{V}}$), and ragged ($\mathcal{R}$) walks. The external input $y \in \Sigma_{drop}$ is omitted below.

Lemma 16.

$$\overline{\mathcal{R}} \circ (\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}) = \emptyset \tag{1}$$

$$\mathcal{R} \circ \overline{(\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}})} = \emptyset \tag{2}$$

Both parts of the latter lemma follow directly from the definition above. We state the lemma in order to reflect the following intuitive property: the ragged walks are the only user walks that can be successfully composed with the invalidated or incomplete walks (part 1), and vice versa (part 2).

Results of this section are used in the correctness proofs of Sect. 5.2. One of our main concerns there is to make sure that after the invalidated and incomplete walks were dropped, no walks from set

$$\mathcal{R} \circ (\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}})$$

are passed as external inputs. These walks (according to the lemma above) will no longer be composable, and if passed, will violate one of the guard conditions of the processor core steps (see Sect. 3.3.4).

3.4.3 TLB Equivalence

In this section we introduce two technical lemmas which in the future help us to reduce the length of the arguments in the correctness proofs significantly. The first lemma asserts that for any step of the original semantics there is a *TLB equivalent* step of the general semantics. Intuitively, two steps are called TLB equivalent if for any given ISA configuration they produce next-state configurations with identical TLB components.

Lemma 17. *Let c be an ISA configuration and let $x \in \Sigma_{tlb}$ be an external input for the step performed in the original semantics. If the corresponding step performed in the general semantics uses input $y \in \Sigma_{add} \cup \Sigma_{drop}$ that satisfies the conditions below, these two steps are TLB equivalent.*

$$\delta_{ISA}(c,x).tlb = \delta_{tlb}(c.tlb,y)$$

For steps which add walks into the TLB the conditions are as follows.

$$\begin{aligned} x = x.(winit,a,guest) &\rightarrow y = (winit, vmid(c) \circ 0_8 \circ x.a, pto(c)) \\ x = x.(winit,a,user) &\rightarrow y = (winit, asid(c) \circ x.a, npto(c)) \\ x = x.(wext,w_1,w_2) &\rightarrow y = (wext, x.w_1, pte(c,x)) \end{aligned}$$

For steps which drop walks from the TLB

$$x = (w_1, w_2, eev)$$

the conditions are as follows.

$$\begin{aligned} /user(c) \wedge invlpg(c,x) &\rightarrow y = (drop, inva(c,x)) \\ guest(c) \wedge flusht(c,x) &\rightarrow y = (drop, vmid(c)) \\ host(c) \wedge flusht(c,x) &\rightarrow y = (drop, all) \end{aligned}$$

Proof of lemma 17. This turns out to be a simple bookkeeping exercise. First we consider addition of the new walks into the TLB. General semantics defines the next state configuration of the TLB component as

$$c'.tlb = c.tlb \cup \{w(y)\}$$

whereas the original semantics defines the next state configuration as

$$c'.tlb = c.tlb \cup \{w(c,x)\}.$$

Thus, to prove the TLB equivalence it suffices to show

$$w(y) = w(c,x).$$

Cases to cover:

- initialization of the guest walk.

$$\begin{aligned} w(y) &= \mathit{initw}(y.\mathit{upa}, y.\mathit{pto}) \\ &= \mathit{initw}(\mathit{vmid}(c) \circ 0_8 \circ x.a, \mathit{pto}(c)) && \text{(assumptions)} \\ &= w(c,x) \end{aligned}$$

- initialization of the user walk.

$$\begin{aligned} w(y) &= \mathit{initw}(y.\mathit{upa}, y.\mathit{pto}) \\ &= \mathit{initw}(\mathit{asid}(c) \circ x.a, \mathit{npto}(c)) && \text{(assumptions)} \\ &= w(c,x) \end{aligned}$$

- extension of the guest walk.

$$\begin{aligned} w(y) &= \mathit{wext}(y.w, y.\mathit{pte}) \\ &= \mathit{wext}(x.w_1, \mathit{pte}(c,x)) && \text{(assumptions)} \\ &= \mathit{wext}(x.w_1, \mathit{pte}(x.w_1, c.m)) \\ &= \mathit{wext}(x.w_1, c.m) && \text{(lemma 15)} \\ &= w(c,x) \end{aligned}$$

- extension of the user walk.

$$\begin{aligned} w(y) &= \mathit{wext}(y.w, y.\mathit{pte}) \\ &= \mathit{wext}(x.w_1, \mathit{pte}(x,c)) && \text{(assumptions)} \\ &= \mathit{wext}(x.w_1, \mathit{pte}(x.w_1 \circ x.w_2, c.m)) \\ &= \mathit{wext}(x.w_1 \circ x.w_2, \mathit{pte}(x.w_1 \circ x.w_2, c.m)) && \text{(equation 4)} \\ &= \mathit{wext}(x.w_1 \circ x.w_2, c.m) && \text{(lemma 15)} \\ &= w(c,x) \end{aligned}$$

In the latter case of the walk extension we use properties of the composition operation. Among others we use that the fault-bits of composition $w_1 \circ w_2$ are identical to the fault-bits of walk w_1 (see p. 56).

$$\begin{aligned} /f(w_1 \circ w_2) &\rightarrow (w_1 \circ w_2).(f_u, f_g) = 0_2 \\ &\rightarrow w_1.(f_u, f_g) = 0_2 \end{aligned} \tag{4}$$

Next we consider dropping of walks from the TLB. Original semantics defines the next state configuration of the TLB depending on type of the invalidating instruction. We split cases according to this type.

- invalidation of a page address. General semantics defines

$$\begin{aligned} c'.\mathit{tlb} &= c.\mathit{tlb} \setminus (\mathcal{I}_{\mathcal{V}}(y) \cup \mathcal{I}_{\mathcal{C}}(y)) \\ &= c.\mathit{tlb} \cap \overline{\mathcal{I}_{\mathcal{V}}(y)} \cap \overline{\mathcal{I}_{\mathcal{C}}(y)}. \end{aligned}$$

For the original semantics we derive

$$\begin{aligned} c'.\mathit{tlb} &= c.\mathit{tlb} \cap \{w \mid w.\mathit{as} = \mathit{inva}(c,x).\mathit{as} \ \rightarrow \ w.\mathit{pa} \neq \mathit{inva}(c,x).\mathit{pa} \wedge w.\ell[0]\} \\ &= c.\mathit{tlb} \cap \{w \mid w.\mathit{upa} \neq \mathit{inva}(c,x) \ \wedge \ (w.\mathit{as} \neq \mathit{inva}(c,x).\mathit{as} \vee w.\ell[0])\}. \end{aligned}$$

To prove the TLB equivalence of two steps we show:

$$\begin{aligned} & c.\mathit{tlb} \cap \overline{\mathcal{I}_{\mathcal{V}}(y)} \cap \overline{\mathcal{I}_{\mathcal{C}}(y)} \\ &= c.\mathit{tlb} \cap \{w \mid w.\mathit{upa} \neq y.\mathit{upa}\} \cap \{w \mid w.\mathit{as} \neq y.\mathit{upa}.\mathit{as} \vee w.\ell[0]\} \\ &= c.\mathit{tlb} \cap \{w \mid w.\mathit{upa} \neq \mathit{inva}(c,x)\} \cap \{w \mid w.\mathit{as} \neq \mathit{inva}(c,x).\mathit{as} \vee w.\ell[0]\} && \text{(assumptions)} \\ &= c.\mathit{tlb} \cap \{w \mid w.\mathit{upa} \neq \mathit{inva}(c,x) \ \wedge \ (w.\mathit{as} \neq \mathit{inva}(c,x).\mathit{as} \vee w.\ell[0])\}. \end{aligned}$$

- invalidation of a virtual machine. General semantics defines

$$\begin{aligned} c'.tlb &= c.tlb \setminus \mathcal{I}_{\mathcal{D}}(y) \\ &= c.tlb \cap \overline{\mathcal{I}_{\mathcal{D}}(y)}. \end{aligned}$$

  For the original semantics we derive

$$\begin{aligned} c'.tlb &= c.tlb \cap \{w \mid w.wm = vmid(c) \to w.pr = 0_8\} \\ &= c.tlb \cap \{w \mid w.wm \neq vmid(c) \ \vee \ w.pr = 0_8\}. \end{aligned}$$

  To prove the TLB equivalence of two steps we show:

$$\begin{aligned} c.tlb \cap \overline{\mathcal{I}_{\mathcal{D}}(y)} &= c.tlb \cap \{w \mid w.upa \notin A_U(y.vm)\} \\ &= c.tlb \cap \{w \mid w.upa \notin A_U(vmid(c))\} && \text{(assumptions)} \\ &= c.tlb \cap \{w \mid w.wm \neq vmid(c) \vee w.pr = 0_8\}. \end{aligned}$$

- invalidation of all translations. The TLB is flushed in both models. □

The second lemma of this section states that steps of the general semantics performed in the given ISA configurations preserve simulation of the TLB component for the next-state configurations. For two ISA configurations c_1 and c_2 we abbreviate

$$sim_{tlb}^{\text{ISA}}(c_1, c_2) \ \leftrightarrow \ c_1.tlb \subseteq c_2.tlb.$$

Lemma 18. *Let c_1 and c_2 be two ISA configurations and let $y \in \Sigma_{add} \cup \Sigma_{drop}$ be an external input used to obtain the next-state configurations c_i' s.t.*

$$c_i'.tlb = \delta_{tlb}(c_i.tlb, y).$$

The simulation of the TLB component is preserved for configurations c_1' and c_2'.

$$sim_{tlb}^{\text{ISA}}(c_1, c_2) \ \to \ sim_{tlb}^{\text{ISA}}(c_1', c_2')$$

Proof of lemma 18. According to the general semantics, there are only two cases:

- addition of walks into the TLB, i.e., $y \in \Sigma_{add}$.

$$\begin{aligned} c_1'.tlb &= c_1.tlb \cup \{w(y)\} \\ &\subseteq c_2.tlb \cup \{w(y)\} && (sim_{tlb}^{\text{ISA}}(c_1, c_2)) \\ &= c_2'.tlb \end{aligned}$$

- dropping of walks from the TLB, i.e., $y \in \Sigma_{drop}$.

$$\begin{aligned} c_1'.tlb &= c_1.tlb \setminus \mathcal{I}(y) \\ &\subseteq c_2.tlb \setminus \mathcal{I}(y) && (sim_{tlb}^{\text{ISA}}(c_1, c_2)) \\ &= c_2'.tlb \end{aligned}$$

□

4 Implementation of Nested MMU

In this chapter we adjust the hardware constructions from [LOP] s.t. after we finish they meet the specifications from Chap. 3. Later on we proceed gradually, but at this point it is important to stress our goal. In order to improve performance, we design the hardware to cache two kinds of walks: i) *hardware guest walks* and ii) *hardware user walks*. In the first case these are the ordinary guest walks from the ISA, exactly as before. However, in the second case these are compositions — as defined in Sect. 3.2.2 — of the ordinary user and guest walks from the ISA. In hardware, at the level of user, the processor core will use the latter compositions directly, after a single look-up in the hardware TLB.

4.1 Redesigning TLB

We start with the presentation of a problem discovered in the process of defining the semantics for machine with NAT. So, consider the following scenario... assume that a certain user address was successfully translated and the corresponding translation is cached by the hardware. From specification of NAT (Sect. 3.2.1) we infer that after the translation process finishes, the hardware TLB contains up to three auxiliary complete guest walks created by the MMU. On the other hand, the software TLB contains — among other translations used to obtain the resulting user walk — software counterparts of those hardware guest walks. We are particularly interested in the one (w_g) which can be composed with the resulting software user walk (w_u) to form a counterpart of the resulting hardware user walk (w_U), i.e.,

$$w_u \circ w_g = w_U .$$

For details we refer to the introductory sections on NAT (Sect. 3.2). Now assume an *invlpg* instruction is executed and its target is the universal page address of that guest walk (w_g). On the level of ISA this does not create any difficulty, we can easily specify the effect of such invalidating transition. But (!) if one drops only the auxiliary guest walks from the TLB (both hardware and software), still there are user walks left, w_U in the hardware and w_u in the software TLB. This leads to very undesired consequences: the hardware user walk (w_U), which still can be used by the processor core, can no longer be reconstructed from the software user walk (w_u) on the ISA level, since the matching guest walk necessary for composition (w_g) was dropped on the *invlpg*. As a result the hardware is able to perform steps which the ISA computation cannot simulate. Of course, such behavior we cannot allow.

So, we change the effect of invalidating transitions in hardware: together with the dropped guest walk we require to drop all (!) matching user walks, to make sure these walks are not used by the hardware. This, however, creates another difficulty we have to overcome. Namely, in hardware we need to invalidate the user walks, which no longer contain the data about the guest walks they were composed from. For the sake of simplicity we consider the model of the

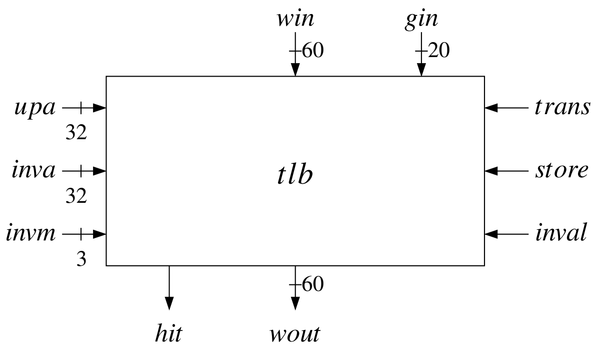


Fig. 14: Interface of TLB for nested translation

hardware TLB where we store all the missing data in dedicated registers[1]. These registers we fill with the *guest page addresses* — page addresses of the guest walk register (w_G) — which we lose after the walk composition.

We spend the following two sections to present both formal specification and hardware implementation of the TLB component. Thus, in Sect. 4.1.1 we describe the changes to the specification of the hardware TLB. The exact implementation of the hardware TLB is then presented in Sect. 4.1.2.

4.1.1 Specification

Since the construction from [LOP] does not provide all features necessary to perform NAT, we propose a slightly more advanced design. Extension of functionality requires changes of the TLB hardware interface. Now it has the following inputs:

- $upa \in \mathbb{B}^{32}$ — universal page address — (virtual) address to look-up in the TLB,
- $win \in \mathbb{B}^{60}$ — walk input — hardware walk (translation) to add into the TLB,
- $gin \in \mathbb{B}^{20}$ — guest input — page address of the guest walk which was used to obtain *win*,
- $inva \in \mathbb{B}^{32}$ — invalidation address — (virtual) address to drop from the TLB,
- $invm \in \mathbb{B}^{3}$ — invalidation mask — control bits to specify the invalidation query,
- $trans \in \mathbb{B}$ — translation request — control signal to execute the translation query,
- $store \in \mathbb{B}$ — store request — control signal to store the input walk (translation),
- $inval \in \mathbb{B}$ — invalidation request — control signal to execute the invalidation query.

Outputs of the hardware TLB remain unchanged:

- $wout \in \mathbb{B}^{60}$ — walk output — hardware walk to be returned to the nested MMU,
- $hit \in \mathbb{B}$ — TLB hit — control signal indicating that the TLB contains a translation for the requested (virtual) address.

The resulting interface is depicted in Fig. 14. As usual, for the hardware TLB to operate properly, several *operating conditions* have to be satisfied. We list these conditions below.

- As before, at most one request is allowed at a time.

$$trans + store + inval \leq 1$$

- Under the store request, (virtual) address of the written walk — as we already mentioned in the previous section — has to be passed as the invalidation address. This requirement is

[1] Without these extra registers the data about the "origin" of user walks can be restored from the data available in hardware only partially. Moreover, certain restrictions on content of the page tables become necessary.

technical and very similar to the one we already had in [LOP]. It simplifies the hardware mechanism which evicts conflicting[2] walks.

$$store \rightarrow inva = win.upa$$

- Under the invalidation request, invalidation mask is expected to have one of the supported values, which in turn correspond to TLB $flush$, $vmflush$ and $invlpg$ resp.

$$inval \rightarrow invm \in \{111, 011, 000\}$$

- One more completely technical condition which simplifies the implementation: on translation and store requests, we expect bits of the invalidation mask to be all zeros.

$$trans \vee store \rightarrow invm = 000$$

In the remainder of this section we formulate all required properties of the hardware TLB component. For convenience we split these properties w.r.t. the request initiated in the current hardware configuration h. As usual, we denote by h' the resulting hardware configuration.

- Formally, the hardware TLB $h.tlb$ consists of three components:

$$\begin{aligned} h.tlb.w &: [1:N] \rightarrow \mathbb{B}^{60} \\ h.tlb.g &: [1:N] \rightarrow \mathbb{B}^{20} \\ h.tlb.v &: [1:N] \rightarrow \mathbb{B} \end{aligned}$$

  where $N \in \mathbb{N}$ denotes the maximum number of simultaneously stored entries. In [LOP] the set of (valid) walks stored in a TLB was defined as

$$tlbset(tlb) = \{tlb.w(i) \mid tlb.v(i) \wedge i \in [1:N]\}.$$

  We refer to the set of walks stored in the hardware TLB of configuration h using the following shorthand.

$$tlbset(h) \equiv tlbset(h.tlb)$$

  In a similar way we abbreviate every definition Z introduced below in this section.

$$Z(h) \equiv Z(h.tlb)$$

- Obviously, in the absence of any request, the content of the hardware TLB (the set of stored walks) does not change.

$$/(trans(h) \vee store(h) \vee inval(h)) \rightarrow tlbset(h') = tlbset(h)$$

- Under a translation request, the content of the hardware TLB does not change either. Moreover, in case of a TLB hit, the output walk matches the (virtual) address requested and, clearly, comes from the TLB.

$$\begin{aligned} trans(h) &\rightarrow tlbset(h') = tlbset(h) \\ trans(h) &\rightarrow (hit(h) \rightarrow wout(h) \in tlbset(h)) \\ trans(h) &\rightarrow (hit(h) \rightarrow wout(h).upa = upa(h)) \end{aligned}$$

- After execution of a store request, the resulting hardware TLB contains the (former) walk written, though it might drop one of the walks stored. Conflicting walk is always evicted, and in case there is none and the hardware TLB is full, the victim walk is dropped.

$$\begin{aligned} store(h) &\rightarrow tlbset(h') \subseteq tlbset(h) \cup \{win(h)\} \\ store(h) &\rightarrow tlbset(h') \cap \{w \mid w.upa = win(h).upa\} = \{win(h)\} \end{aligned}$$

[2] We call two walks in the hardware TLB *conflicting* if they provide translation for the identical (virtual) addresses. It is crucial to eliminate conflicting walks and keep translations in the hardware TLB "uniquely" present.

- Finally, after execution of the invalidation request, the resulting hardware TLB does not contain the specified set of walks. For simplicity we call the latter set *invalidated*, and define it formally as

$$invset(h) = \begin{cases} tlbset(h) & invm(h) = 111 \\ tlbset(h) \cap \{w \mid w.upa \in A_U(inva(h).vm)\} & invm(h) = 011 \\ tlbset(h) \cap \{w \mid w.upa = inva(h)\} & invm(h) = 000. \end{cases}$$

In addition, according to the semantics of *invlpg* from Sect. 3.3.6, the hardware TLB loses all incomplete walks from the target address space, as well as all user walks matching the invalidated (guest) walk. We call the set of incomplete walks to be dropped *incset* and define it as

$$incset(h) = \begin{cases} tlbset(h) \cap \{w \mid (w.as = inva(h).as) \wedge /w.\ell[0]\} & invm(h) = 000 \\ \emptyset & \text{otherwise.} \end{cases}$$

We call the other set of walks to be dropped *ragset*[3]. In order to formalize the set of ragged (hardware) walks, we require the following technical definition.

Definition 6 (guest page address). *For walks $w \in tlbset(h)$:*

$$gpa(w,h) \;\equiv\; \varepsilon\{h.tlb.g(i) \mid h.tlb.v(i) \wedge (h.tlb.w(i) = w)\}$$

The guest page address of walk w from the hardware TLB $h.tlb$ is stored in the component $h.tlb.g(i)$ if walk w is stored in the entry i. The definition above is well-defined due to the invariant 2, which preserves uniqueness of the walks stored in hardware.

Formally, we define the *ragset* as

$$ragset(h) = \begin{cases} tlbset(h) \cap \{w \mid (w.upa \in A_U(inva(h).vm)) \wedge & inva(h) \in A_G \wedge \\ \qquad\qquad ((gpa(w,h) = inva(h).pa) \vee w.f_g)\} & invm(h) = 000 \\ \emptyset & \text{otherwise.} \end{cases}$$

Note, since the incomplete guest walks are automatically dropped on *invlpg*, we also include into the set of ragged walks all *guest faulty* (see Sect. 3.2.1) user walks, which we identify using flag $w.f_g$, from the given address space. Now, the effect of the invalidation request on the hardware TLB is obvious.

$$inval(h) \rightarrow tlbset(h') = tlbset(h) \setminus (invset(h) \cup incset(h) \cup ragset(h))$$

This finishes the hardware specification of the TLB component. Easy to see that in this specification the TLB is functional w.r.t. addresses of the stored walks. To reflect the latter property we introduce an invariant.

Invariant 2. *For walks $w_1, w_2 \in tlbset(h)$:*

$$w_1.upa = w_2.upa \rightarrow w_1 = w_2$$

In the next section we implement the TLB component according to the specification above. Note that targeting the latter specification, we present a construction that can serve at most one request at a time.[4]

4.1.2 Construction

In a nutshell, construction remains almost the same as in [LOP]: still it is a cache for address translations (walks). However, it acquires several important properties which are absolutely

[3] This name is motivated by the situation which occurs in software: after execution of the *invlpg* on the level of ISA, the user walks matching the invalidated guest walk cannot be composed with the walks remaining (see Sect. 3.4.2).

[4] The first operating condition restricts inputs of the hardware TLB appropriately. Of course, the latter operating condition can be relaxed with a slightly more advanced implementation, but we do not bother.

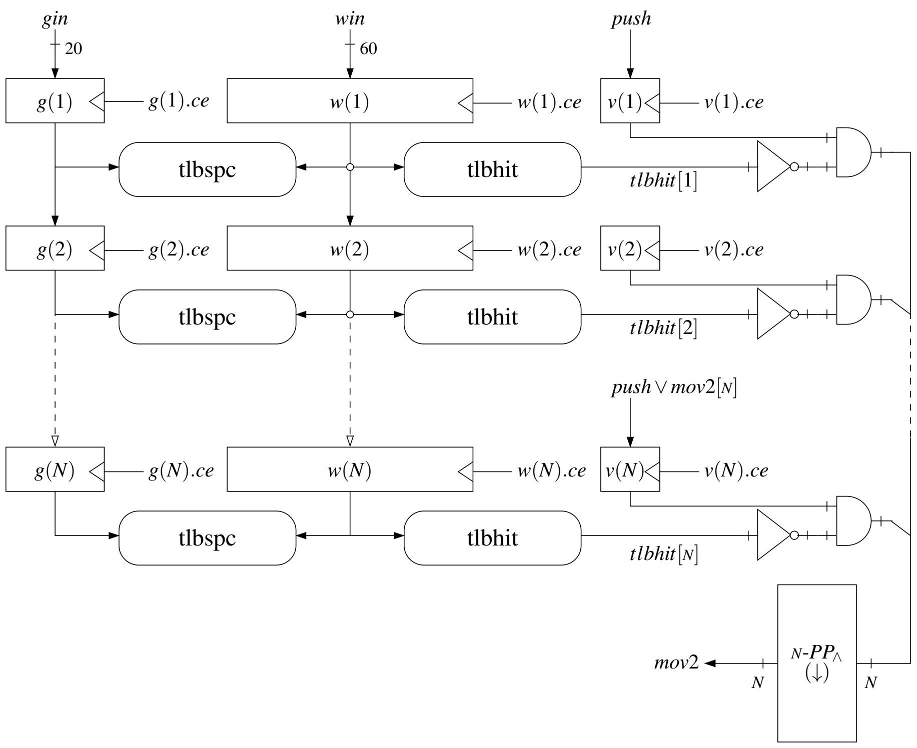


Fig. 15: Basic construction of the hardware TLB

necessary for the nested translation. Thus, our construction allows to perform invalidation of multiple translations and — despite that — use the hardware cache efficiently. First, we present a basic construction of the novel hardware TLB. Then, we extend that basic construction in two more steps:

i) we design a circuitry which performs dropping of all hardware ragged walks (user walks composed from the guest walk being invalidated), and
ii) we add support for invalidation of (all translations of) a particular VM.

As an intermediate signal in the forthcoming hardware constructions we use

$$tlba(h) = \begin{cases} upa(h) & trans(h) \\ inva(h) & \text{otherwise.} \end{cases}$$

The reason being is that we tend to reuse the same circuits when computing internal hardware hits, which in turn depend on the translation (*upa*) and invalidation (*inva*) addresses in exactly the same way.

Basic Design

The basic construction of the hardware TLB is depicted in Fig. 15. The outputs, which are not shown in the figure, are computed in the usual way by the OR-trees.

$$\begin{aligned} hit(h) &= \bigvee_i \; tlbhit(h)[i] \wedge trans(h) \\ wout(h) &= \bigvee_i \; tlbhit(h)[i] \wedge h.tlb.w(i) \end{aligned}$$

There are circuits for computation of hardware *hit* and *incomplete* signals on per entry basis, i.e., for $i \in [1 : N]$: “tlbhit” and “tlbspc” resp. Construction of these circuits can be easily deduced from the formal specification of their outputs. Using the intermediate signals

$$\begin{aligned} hit(h)[i] &\equiv h.tlb.w(i).upa = tlba(h) \\ inc(h)[i] &\equiv (h.tlb.w(i).as = tlba(h).as) \wedge /h.tlb.w(i).\ell[0] \end{aligned}$$

we define:

$$\begin{aligned} tlbhit(h)[i] &\equiv h.tlb.v(i) \wedge ((invm(h) = 111) \vee hit(h)[i]) \\ tlbspc(h)[i] &\equiv h.tlb.v(i) \wedge ((invm(h) = 000) \wedge inc(h)[i]). \end{aligned}$$

Note, the definitions above are temporary and serve to present a simplified version of the design. The final definitions of the latter signals are made later in this section, when we describe dropping of the virtual machines and ragged hardware walks. We define two more auxiliary signals in order to simplify the presentation below:

- a *push* (into entry $i = 1$) — indicates that data are written into the hardware TLB,
- a *move-to* (entry $i > 1$) — indicates that data are "moved" from the entry $(i-1)$ into the entry i.

$$\begin{aligned} push(h) &\equiv store(h) \\ mov2(h)[i] &\equiv \bigwedge_{j<i} h.tlb.v(j) \wedge /tlbhit(h)[j] \end{aligned}$$

Using these auxiliary signals, we easily define the inputs of hardware *walk* and *guest address* registers (on the left, $x \in \{w, g\}$)

$$\begin{aligned} i = 1: &\qquad h.tlb.x(i).in = xin(h) \\ i > 1: &\qquad h.tlb.x(i).in = h.tlb.x(i-1) \end{aligned}$$

and *valid bits* (on the right).

$$\begin{aligned} i = 1: &\qquad h.tlb.v(i).in \equiv push(h) \\ i > 1: &\qquad h.tlb.v(i).in \equiv push(h) \wedge mov2(h)[i] \end{aligned}$$

Finally, for the clock-enable signals we split cases according to the initiated request. Note, there is no sense to clock the data register if input of the corresponding valid bit differs from one.

$$\begin{aligned} h.tlb.v(i).ce &\equiv \begin{cases} tlbhit(h)[i] \vee h.tlb.v(i).in & store(h) \\ tlbhit(h)[i] \vee tlbspc(h)[i] & inval(h) \end{cases} \\ h.tlb.x(i).ce &\equiv h.tlb.v(i).in \end{aligned}$$

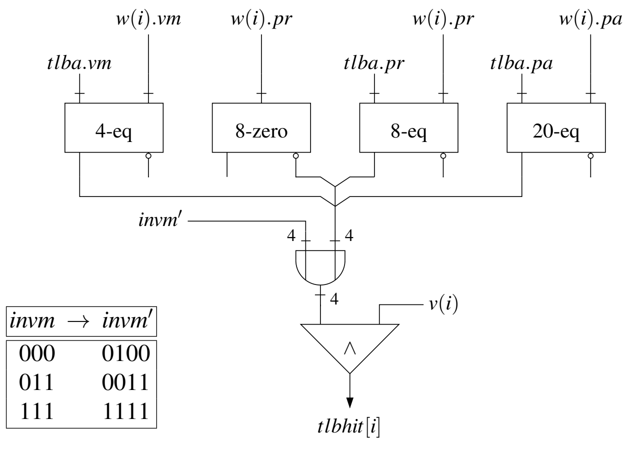
Clearly, this basic construction refines the one presented in [LOP]. It provides an equivalent functionality, however supports extensions unavailable for our prior construction. This property comes with the price that whenever the hardware TLB fills densely enough, a significant portion of the registers is overwritten at once (!) on every store request.

Dropping Virtual Machines

First we extend the basic TLB design to support the overloaded $flusht$ instruction, which executed at the level of guest flushes all TLB translations of the running VM (see Sect. 3.3.6). Recall, we select both the walk to be output on translation request and the walks to be invalidated on invalidation request using the internal hardware hit signal. Also recall, on translation requests the invalidation mask is guaranteed to be all zeros (by the operating condition, see Sect. 4.1.1), whereas on invalidation requests the mask is used to specify the invalidation query further. So, in order to fulfill the hardware specifications, it suffices simply to adjust the former computation of the $tlbhit(h)$ signal appropriately.

$$tlbhit(h)[i] \equiv h.tlb.v(i) \wedge \begin{cases} h.tlb.w(i).upa = tlba(h) & invm(h) = 000 \\ h.tlb.w(i).upa \in A_U(tlba(h).vm) & invm(h) = 011 \\ 1 & invm(h) = 111 \end{cases}$$

There are no changes in handling of the translation requests. The resulting hardware is completely trivial and depicted in Fig. 16.



$invm$	$\rightarrow$ $invm'$
000	0100
011	0011
111	1111

Fig. 16: Circuit “tlbhit”

Dropping Ragged Walks

As the second extension we introduce a mechanism for dropping all the ragged walks from the hardware TLB. In specifications of the hardware TLB (see definition of the $ragset(h)$ in Sect. 4.1.1) we require to drop all the user walks obtained through any matching guest walk whose page address is invalidated. We can easily determine the ragged walk in the hardware TLB by matching its guest page address (gpa) with the page address of the invalidation address ($tlba(h)$). In particular, on per entry basis (i.e., for $i \in [1:N]$) we deliberately compute the following hardware signal.

$$\begin{aligned} rag(h)[i] \equiv (tlba(h).pr = 0_8) \wedge (h.tlb.w(i).upa \in A_U(tlba(h).vm)) \wedge \\ (\ (h.tlb.g(i) = tlba(h).pa) \vee h.tlb.w(i).f_g\ ) \end{aligned}$$

This signal can already be used to drop the ragged hardware walks on invalidation request ($inval(h)$). However, we want to reuse most of the definitions made in the previous section and change the basic design as little as possible. We notice that the ragged walks are in certain sense “incomplete” without the invalidated walks... Since on $inval(h)$ all incomplete walks are necessary dropped, we adjust the former computation of $tlbspc(h)$ signal s.t. it also includes the ragged walks to be dropped.

$$tlbspc(h)[i] \equiv h.tlb.v(i) \wedge (invm(h) = 000) \wedge (rag(h)[i] \vee inc(h)[i])$$

The corresponding circuit is depicted in Fig. 17. In the dotted boxes we highlight the hardware parts which can be saved, i.e., their outputs can be taken from the hardware for computation of the hit signal ($tlbhit(h)$).

Implementation Correctness

In the following trivial lemma we encapsulate the correctness of the TLB implementation.

Lemma 19. *Assume that the invalidation request signal ($inval(h)$) is active.*

$$invset(h) \subseteq \{h.tlb.w(i) \mid tlbhit(h)[i]\} \tag{1}$$

$$ragset(h) \cup incset(h) \subseteq \{h.tlb.w(i) \mid tlbspc(h)[i]\} \tag{2}$$

Proof of lemma 19.1. By case split on the value of the invalidation mask:

- $invm(h) = 111$. Dropping all walks.

$$\begin{aligned} invset(h) &= tlbset(h) \\ &= \{h.tlb.w(i) \mid h.tlb.v(i)\} \\ &= \{h.tlb.w(i) \mid tlbhit(h)[i]\} \end{aligned}$$

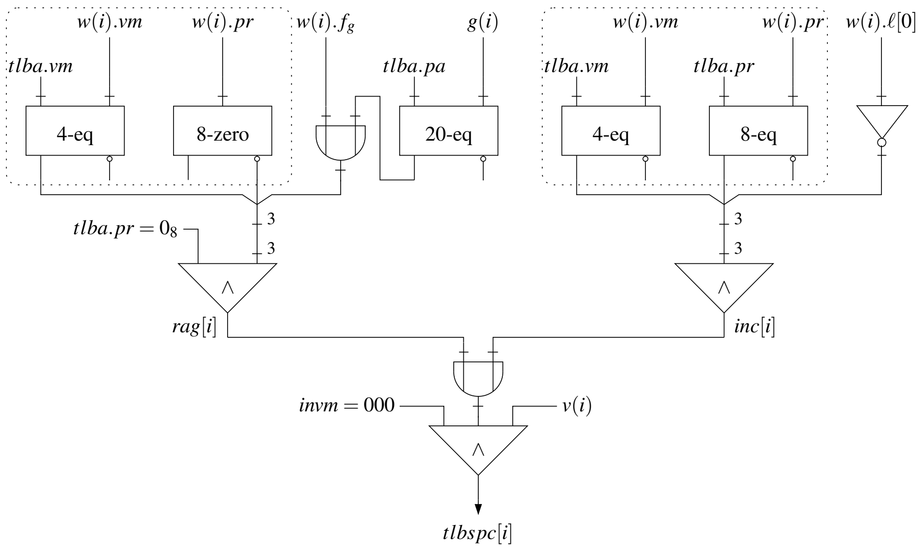


Fig. 17: Circuit "tlbspc"

- $invm(h) = 011$. Dropping virtual machines.

$$\begin{aligned} invset(h) &= tlbset(h) \cap \{w \mid w.upa \in A_U(inva(h).vm)\} \\ &= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge (h.tlb.w(i).upa \in A_U(inva(h).vm))\} \\ &= \{h.tlb.w(i) \mid tlbhit(h)[i]\} \end{aligned}$$

- $invm(h) = 000$. Dropping individual walks.

$$\begin{aligned} invset(h) &= tlbset(h) \cap \{w \mid w.upa = inva(h)\} \\ &= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge (h.tlb.w(i).upa = inva(h))\} \\ &= \{h.tlb.w(i) \mid tlbhit(h)[i]\} \end{aligned}$$ □

Proof of lemma 19.2. Assume we have

$$invm(h) = 000$$

since otherwise there is nothing to show. For the incomplete walks we argue:

$$\begin{aligned} incset(h) &= tlbset(h) \cap \{w \mid (w.as = inva(h).as) \wedge /w.\ell[0]\} \\ &= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge (h.tlb.w(i).as = inva(h).as) \wedge /h.tlb.w(i).\ell[0]\} \\ &= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge inc(h)[i]\}. \end{aligned}$$

For the ragged walks we assume also

$$inva(h).pr = 0_8$$

since otherwise there is nothing to show. Moreover, for the guest page address we show:

$$h.tlb.v(i) \ \rightarrow \ gpa(h.tlb.w(i), h) = h.tlb.g(i).$$

Clearly we have

$$h.tlb.v(i) \ \rightarrow \ h.tlb.w(i) \in tlbset(h).$$

Using the definition of the guest page address and invariant 2 we easily obtain

$$\begin{aligned} gpa(h.tlb.w(i), h) &= \varepsilon\{h.tlb.g(j) \mid h.tlb.v(j) \wedge (h.tlb.w(j) = h.tlb.w(i))\} \\ &= \bigvee_j \ (\ h.tlb.g(j) \wedge h.tlb.v(j) \wedge (h.tlb.w(j) = h.tlb.w(i))\ ) \\ &= h.tlb.g(i) \wedge h.tlb.v(i). \end{aligned}$$

Given the lines above we analogously argue:

$$\begin{aligned}
ragset(h) &= tlbset(h) \cap \{w \mid (w.upa \in A_U(inva(h).vm)) \wedge \\
&\qquad\qquad ((gpa(w,h) = inva(h).pa) \vee w.f_g)\} \\
&= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge (h.tlb.w(i).upa \in A_U(inva(h).vm)) \wedge \\
&\qquad\qquad ((h.tlb.g(i) = inva(h).pa) \vee h.tlb.w(i).f_g)\} \\
&= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge rag(h)[i]\}.
\end{aligned}$$

Collecting the arguments we conclude:

$$\begin{aligned}
ragset(h) \cup incset(h) &= \{h.tlb.w(i) \mid h.tlb.v(i) \wedge (rag(h)[i] \vee inc(h)[i])\} \\
&= \{h.tlb.w(i) \mid tlbspc(h)[i]\}.
\end{aligned}$$ □

4.2 Redesigning MMU

Of course, the major changes in hardware concern implementation of the memory management unit — the component responsible for address translation — for the nested translation. Since we tend to reuse the constructions developed previously in [LOP], out main challenge here is to find a representation of the nested translation scheme s.t. the hardware we developed for the address translation previously can be used as part of that scheme. We proceed as follows:

i) in Sect. 4.2.1 we identify those portions of hardware which can be reused, and explain how to construct a *nested* MMU out of those pieces, then
ii) in Sect. 4.2.2 we implement the nested MMU in hardware, and finally
iii) in Sect. 4.3 we prove that our implementation is actually live.

4.2.1 Specification

Since in prospect we plan to replace the original MMU component from [LOP] by the nested MMU developed here, we clearly want to keep the old interfaces unchanged, unless changes are unavoidable. Thus, as the first and the most obvious step we add only two more inputs to the original MMU interface, see Fig. 18:

- $pto_U \in \mathbb{B}^{20}$ — one more page table origin, for the nested address translation, and
- $vmflush \in \mathbb{B}$ — control signal, to invalidate the entire VM (implements $flusht$ instruction executed at the level of guest). The VM to be invalidated is chosen using the top-most four bits of the *inva* input.

As another obvious step we change the original MMU component to work with the universal addressing. This simple change can be described as follows: i) the width of address to be translated increases by 12 bits, and ii) all internal MMU circuits adjust to fit the new address width s.t. changes propagate through the entire design. Taking these modifications into account, we finish specification of the MMU interfaces.

In addition to the user page table origin (pto_U) we specified above, the translation interface includes the following inputs and outputs:

- input $treq \in \mathbb{B}$ — translation request — control signal to start the translation,
- input $upa \in \mathbb{B}^{32}$ — universal page address — virtual address to be translated,
- input $pto_G \in \mathbb{B}^{20}$ — page table origin — page address of the root page table to be used for translation,
- output $busy \in \mathbb{B}$ — MMU busy — control signal indicating that the MMU is currently translating and its output is not yet meaningful, and
- output $wout \in \mathbb{B}^{60}$ — walk output — walk returned for the virtual address (*upa*).

The invalidation interface comprises the aforementioned special input $vmflush$ and all signals from the corresponding interface of the simple MMU:

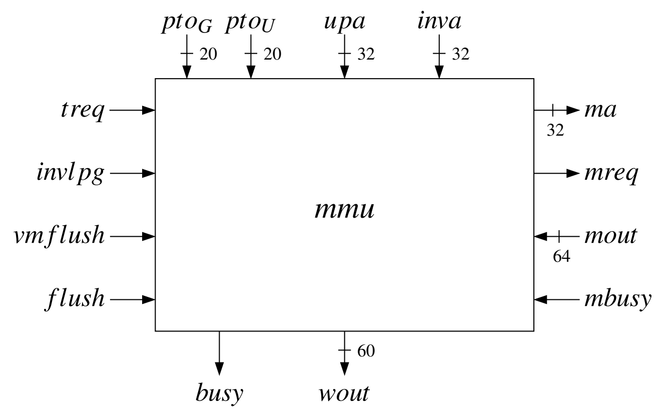


Fig. 18: Interface of MMU for nested translation

- input $flush \in \mathbb{B}$ — flush — control signal to drop all stored translations,
- input $invlpg \in \mathbb{B}$ — invalidate page — control signal to drop a translation, and
- input $inva \in \mathbb{B}^{30}$ — invalidation address — universal address of translations to be dropped.

The memory interface of the nested MMU repeats the corresponding interface of the simple MMU. For details we refer to [LOP].

Just as for the TLB component above, we collect all properties of the nested MMU component. Overall, most properties of the original MMU repeat for the nested MMU. For compatibility with the original MMU we use the same naming as we used for the common interface signals.

- Formally, we define the nested MMU $h.mmu$ as follows[5]:

$$\begin{aligned} h.mmu.tlb &\in [1:N] \to K_{uwalk} \times \mathbb{B}^{20} \times \mathbb{B} \\ h.mmu.w_G &\in K_{uwalk} \\ h.mmu.w_U &\in K_{uwalk}. \end{aligned}$$

- the MMU component is busy in hardware configuration h if and only if i) there is a translation request in configuration h and ii) there is no hit for the given universal page address in the hardware TLB of configuration h.

$$busy(h) \leftrightarrow treq(h) \wedge /tlb.hit(h)$$

  Later we show that the MMU component is busy in hardware configuration h if and only if either i) the internal walk register (w_U) is being initialized, or ii) a memory access takes place, or iii) a so-called nested translation access takes place, or iv) the hardware TLB is being written in configuration h.

- On a hit in the hardware TLB in hardware configuration h, the walk output of the MMU component in configuration h matches the given universal page address. Moreover, the latter walk is either faulty, or complete, or both. Therefore, it can be used for translated access.

$$\begin{aligned} tlb.hit(h) &\to wout(h).upa = upa(h) \\ tlb.hit(h) &\to f(wout(h)) \vee wout(h).\ell[0] \end{aligned}$$

- We finish with specification of the invalidation interface of the nested MMU. The following lines are self-explaining, h and h' below denote resp. the current and the resulting hardware configurations.

[5] As we already mentioned before (in Sect. 4.1.1), in hardware we additionally store with every user walk a page address of the guest walk through which the user walk was obtained. Details of this we cover later, in Sect. 4.2.2.

$$\begin{aligned} flush(h) &\rightarrow tlbset(h') = \emptyset \\ vmflush(h) &\rightarrow tlbset(h') = tlbset(h) \setminus \{w \mid w.upa \in A_U(inva(h).vm)\} \\ invlpg(h) &\rightarrow tlbset(h') = tlbset(h) \setminus \{w \mid (w.upa = inva(h)) \vee \\ &\qquad\qquad (w.as = inva(h).as) \wedge \overline{w.\ell[0]} \vee \\ &\qquad\qquad (inva(h) \in A_G \rightarrow rag(w,h))\} \end{aligned}$$

In the last line above we used the following abbreviation.

$$rag(w,h) \equiv (w.upa \in A_U(inva(h).vm)) \wedge ((gpa(w,h) = inva(h).pa) \vee w.f_g)$$

The resulting construction we call an MMU for the nested translation or *nested MMU* for simplicity. In the remainder of this chapter we consider only the nested MMU(s), so often we drop the word nested and simply use the name “MMU” to refer to this construction.

4.2.2 Construction

To start the section we describe how to construct the nested MMU in a rather sketchy fashion. In a nutshell, the nested MMU consists of the following sub-components:

- *one* hardware TLB for universal addressing — $mmu.tlb$, and
- *two* universal walk registers — $mmu.w_G$ and $mmu.w_U$.

Its overall (very general) construction scheme is depicted in Fig. 19. Conceptually, the nested MMU component there is split into two parts. The first part (on the left) is responsible for translation of the user addresses, whereas the second part (on the right) — for translation of the guest addresses. Each part contains i) a walking unit (see below), which is directly connected to ii) a universal walk register, and iii) a small circuit (a multiplexer) computing the address of the page table entry to be accessed within the process walk extension. Except that, the left part contains another small circuit for computation of the walk composition. Since, the same sub-components occur at both sides, we naturally introduce the following indexing: names of all components — and signals they generate — occurring on the left we index using capital U, whereas names of components on the right are indexed using capital G. Note that the TLB component is accessed whenever necessary by both sides (in turns), but does not belong to either of them.

For technical reasons we distinguish between the requests for translation of the user addresses and the requests for translation of the guest addresses. The former ones we call *nested* translation requests, whereas the latter ones — *simple* translation requests.

$$\begin{aligned} nreq(h) &\equiv treq(h) \wedge (upa(h) \in A_U) \\ sreq(h) &\equiv treq(h) \wedge (upa(h) \in A_G) \end{aligned}$$

Clearly, the left hand side of the MMU component is designed to handle the nested translation requests, whereas the right hand side — to handle the simple translation requests. In case of a nested translation request and in the absence of hits in the hardware TLB — after initialization of the *user walk register* ($mmu.w_U$) — another translation request is initiated: a simple translation request to the right hand side of the MMU component. At that point the regular translation scheme is used: in the absence of hits in the hardware TLB, the *guest walk register* ($mmu.w_G$) is initialized and the ordinary translation starts. When the latter translation ends, the internal translation call returns, which in turn completes one step of the nested walk extension.

Next in this section we cover the details of exact implementation of the nested MMU in hardware. To be clear, in our hardware implementation we pursue the following goal: in the absence of nested translation requests we want the nested MMU to behave exactly (!) like the simple MMU[6]. Taking into account that control over the simple MMU was performed via the component's internal state machine, we decide to include the latter machine into the nested MMU, as a separate sub-component, entirely and without changes. Novel capabilities of the nested

[6] This will simplify the future correctness proof considerably.

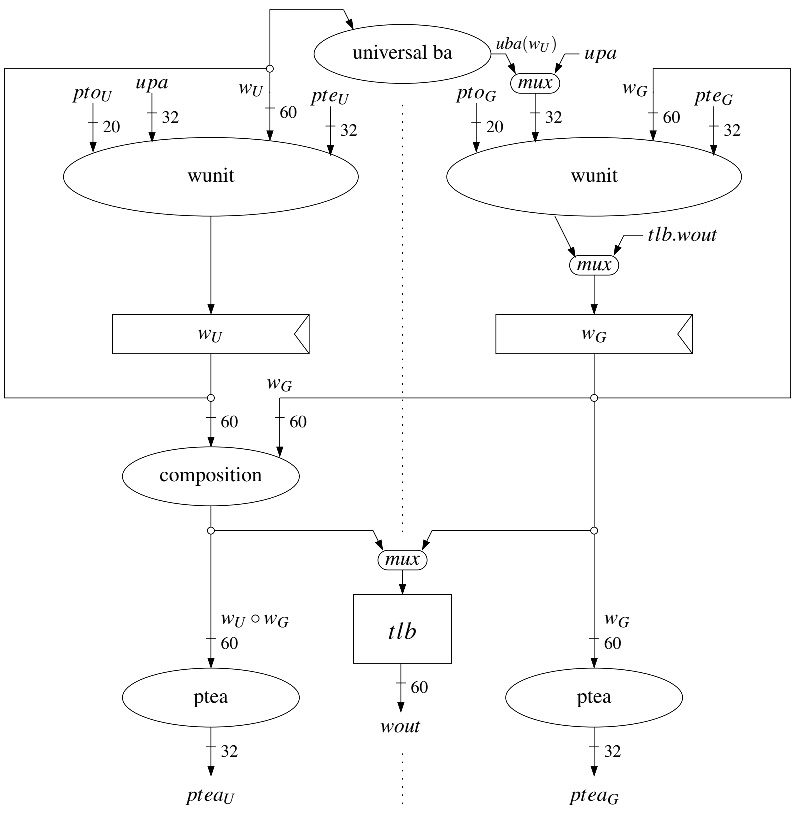


Fig. 19: Hardware layout of the MMU for nested translation. To distinguish between the addresses used by two components we additionally suffixed the *ptea* in the construction above. For the simple MMU we denoted the address by $ptea_G$, since in the first case we extend the guest walk (w_G). For the nested MMU, though we actually extend composition of the user walk (w_U) and the current guest walk (w_G), we denoted the address by $ptea_U$ for simplicity.

MMU are required only in case of nested translations. Therefore, in case of the simple translation requests, the control automaton (CA) of the nested MMU should delegate these requests to the CA of the simple MMU and remain in its idle state. As simple as that. We structure the content of this section as follows:

- we start with the presentation of the control automaton (of the nested MMU),
- then, we describe the implementation of the component's data paths, and
- in the end, we connect interfaces of the involved sub-components and memory.

Before we proceed to definitions, we discuss the notation we use below. First, in the scope of this section all signals are defined w.r.t. the current hardware configuration. So, in order to make the forthcoming definitions "lighter", we drop the h, meaning that all signals below belong to the same hardware configuration. For instance, the predicate $nreq(h)$, which we defined above, can be rewritten simply as

$$nreq \equiv treq \wedge (upa \in A_U).$$

Also, all signals which are used in the short form (without mentioning the component they belong to) are implicitly treated to belong to the nested MMU. In the definition above the *upa* is meant to be the input address of the nested MMU, i.e., *mmu.upa*. Signals belonging, for instance, to the TLB sub-component we distinguish by using the long form, i.e., simply by prefixing them with *tlb*.

For convenience we introduce a shorthand to abbreviate the state of the nested MMU in which both its control automata reside in their idle states.

$$idle \equiv idle_U \wedge idle_G$$

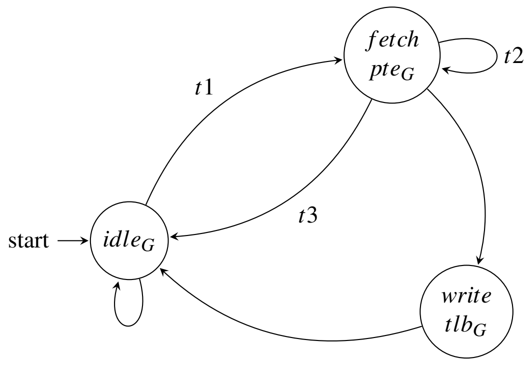


Fig. 20: Control automaton of the nested MMU for simple translation

Before we proceed with implementation of the nested MMU, we add a mechanism to restore the latter state after losses of the translation requests (due to interrupts). For that we add one more control flag (*abort*), which we implement as an ordinary set-clear flip-flop.

$$\begin{aligned} abort.set &= \overline{abort} \wedge \overline{idle} \wedge \overline{treq} \\ abort.clr &= abort \wedge idle \vee reset \end{aligned}$$

Control Automaton of Simple MMU

The simple control automaton is depicted in Fig. 20. Exactly this construction was used in [LOP] to control the (simple) MMU. For convenience we keep the names for control stages and labels for transitions original. Note, using suffix G we distinguish control states of the simple automaton from control states of the nested automaton, which we denote using suffix U. The purpose of the simple automaton also remains unchanged: within the nested MMU it controls the simple translation calls (in state *fetch-pte*) and stores the acquired simple translations in the TLB (state *write-tlb*).

We explain the transitions next. Though we mention certain things to improve the presentation, here we define only (!) conditions when a particular transition is taken. Everything else we define later in dedicated sections.

- The nested CA, which now "wraps" the simple CA, initiates a simple translation sub-request in two cases: i) as the nested translation call in state *nested-call*, and ii) if the simple address translation is performed (at the level of guest). In the presence of the *abort* signal, the request for the simple translation is raised only if both control automata reside in their idle states (*idle*).

  $$treq_G \equiv (nested\text{-}call \wedge nreq \vee sreq) \wedge (abort \rightarrow idle)$$

  Of course, we could simply wait for the absence of the *abort*, but that would take an extra hardware cycle. Instead, we raise the simple translation request as soon as the *abort* clear signal is active.

  In the absence of *reset*, the automaton leaves state $idle_G$ only if a simple translation request leads to a miss in the hardware TLB.

  $$(t1) \equiv idle_G \wedge /reset \wedge treq_G \wedge /tlb.hit$$

- In state *fetch-pte*$_G$ — as the state's name suggests — the memory is accessed in order to read the page tables. In total there are three transitions leaving state *fetch-pte*$_G$. The first one ($t2$) is a self-loop. It implements the process of busy waiting for the memory access to finish. The second transition ($t3$) is, obviously, included to make sure that the control automaton is initialized properly on reset and in order to return control to state $idle_G$ after abortions.

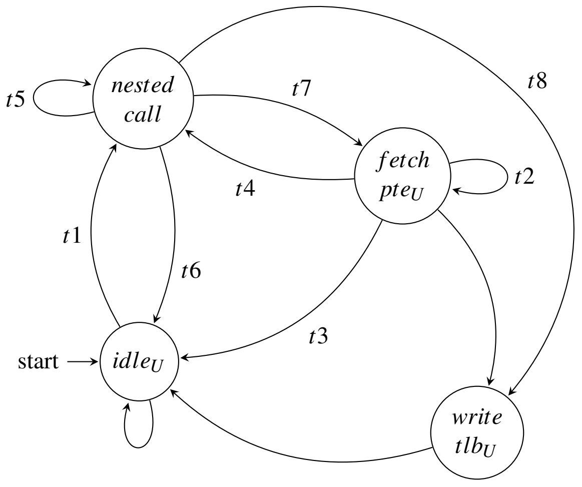


Fig. 21: Control automaton of the nested MMU for nested translation

$$(t2) \equiv \textit{fetch-pte}_G \wedge /\mathit{reset} \wedge \mathit{mbusy}$$
$$(t3) \equiv \textit{fetch-pte}_G \wedge (\mathit{reset} \vee /\mathit{mbusy} \wedge /\mathit{treq}_G)$$

Definitions of transitions $t2$ and $t3$ are self-explaining. The remaining transition (nameless one) is taken "otherwise", i.e., if none of the transitions above is taken. In the latter case, control is brought to state *write-tlb*$_G$, where a faulty or complete walk (w_G) is written into the hardware TLB.

- Finally, there is only one transition coming out of state *write-tlb*$_G$. It returns control to state $idle_G$ while the hardware TLB is being written. Every process of simple translation which was not aborted terminates via this transition.

Control Automaton of Nested MMU

Since internally the nested MMU is more complex than the simple MMU, it has more complicated control mechanism (as can be spotted in Fig. 21). The automaton there has one state more compared to the control automaton of the simple MMU. We have to account for one more state (*nested-call* in the figure) to implement the nested translation calls. These calls are used for translation of the (virtual) base addresses of the user walk (w_U), since — as it might be concluded already — in the walk registers we store the hardware counterparts of the specification user (w_u) and guest (w_g) walks. The remaining states acquire the meanings of those states (of the simple MMU) whose names they repeat. In short: state *fetch-pte*$_U$ is used for reading page tables and state *write-tlb*$_U$ is, obviously, used for writing the result of nested translation into the TLB. Transitions of the nested automaton are described below.

- Clearly, under the nested translation request, one should first check if the hardware TLB contains a walk which translates the requested address (upa). Same as the $treq_G$, in the presence of the $abort$ signal, the request for the nested translation is raised only if both control automata reside in their idle states ($idle$). To end-up with more uniform notation, we introduce a corresponding shorthand.

$$treq_U \equiv nreq \wedge (abort \rightarrow idle)$$

So, in the absence of $reset$, the automaton leaves state $idle_U$ only if a nested translation request leads to a miss in the hardware TLB.

$$(t1) \equiv idle_U \wedge /reset \wedge treq_U \wedge /tlb.hit$$

- Four transitions in total leave state *fetch-pte*$_U$. The first two are analogous to the corresponding transitions in the simple control automaton. Transition number three ($t4$) brings control to the new state, *nested-call*, where translation of the current base address of user walk (w_U) is requested. Below, the first three transitions are specified formally.

  $$(t2) \equiv \textit{fetch-pte}_U \wedge /\mathit{reset} \wedge \mathit{mbusy}$$
  $$(t3) \equiv \textit{fetch-pte}_U \wedge (\mathit{reset} \vee /\mathit{mbusy} \wedge /\mathit{treq}_U)$$
  $$(t4) \equiv \textit{fetch-pte}_U \wedge /\mathit{reset} \wedge /\mathit{mbusy} \wedge \mathit{treq}_U \wedge /f(\mathit{wext}_U)$$

  Transition $t4$ is taken in the absence of *reset*, only if the result of walk extension (*wext*) is not faulty. The remaining transition (nameless) is taken “otherwise”, i.e., if none of the transitions above is taken. Intuitively, it aborts the process of nested translation on a page fault. In the latter case, control is brought to state *write-tlb*$_U$, where a faulty (in this case) composed walk ($w_U \circ w_G$) is written into the hardware TLB.
- In turn, in state *nested-call* a nested translation request to CA of the simple MMU (simple CA) is initiated. Again, there are four transitions in total coming out of state *nested-call*. And again, the first transition ($t5$) is a self-loop for busy-waiting, but this time one waits for the simple CA to finish its operation. Similarly to transition $t3$ above, the second transition ($t6$) is technical. It brings control back to state idle_U on reset. The third transition ($t7$) here returns control to state *fetch-pte*$_U$, where not faulty and yet incomplete user walk (w_U) is extended. Formally, the first three transitions are specified below.

  $$(t5) \equiv \textit{nested-call} \wedge /\mathit{reset} \wedge \mathit{busy}_G$$
  $$(t6) \equiv \textit{nested-call} \wedge (\mathit{reset} \vee /\mathit{busy}_G \wedge /\mathit{treq}_U)$$
  $$(t7) \equiv \textit{nested-call} \wedge /\mathit{reset} \wedge /\mathit{busy}_G \wedge \mathit{treq}_U \wedge /f(w_U, \mathit{tlb.wout}) \wedge /w_U.\ell[0]$$

  Recall, in Sect. 3.2.1 we defined a pair of walks to be faulting as follows.

  $$f(w_u, w_g) \;\equiv\; f(w_u) \vee f(w_g) \vee f(w_g \mid w_u)$$

  Again, definitions of $t5$ and $t6$ are self-explaining. Transition $t7$ is taken in the absence of *reset* (as usual), only if the result of nested translation (*tlb.wout*) is not faulting (see Sect. 3.3.3) and, moreover, the user walk (w_U) is incomplete. “Otherwise” (if none of the transition above is taken), the remaining transition (nameless one) brings control to state *write-tlb*$_U$. This happens in two cases: either i) the user walk (w_U) and the resulting guest walk (*tlb.wout*) are faulting, or ii) otherwise, simply when the user walk (w_U) is already complete. In the first case the nested translation “fails”, and a faulty composed walk ($w_U \circ w_G$) is written into the hardware TLB, whereas in the second case the nested translation “succeeds”, and a not-faulty composed walk ($w_U \circ w_G$) is written into the hardware TLB. Though it is not strictly necessary, for convenience we give a name to the fourth transition in this case and specify it formally as follows.

  $$(t8) \;\equiv\; \textit{nested-call} \wedge /\mathit{reset} \wedge /\mathit{busy}_G \wedge \mathit{treq}_U \wedge (f(w_U, \mathit{tlb.wout}) \vee w_U.\ell[0])$$

- Analogous to the simple control automaton, one transition is coming out of state *write-tlb*$_U$. It returns control to state idle_U while the hardware TLB is being written. Every process of nested translation which was not aborted terminates via this transition.

Walking Units

Walking unit of the simple MMU is a basis for construction of the *nested* walking unit, i.e., walking unit of the nested MMU. The only difference is, literally, in the modified circuits for computation of the walk initialization and extension functions. The resulting hardware is schematically depicted in Fig. 22. Note that widths of the inputs and outputs changed as well according to the new definitions (see Sect. 3.3.1).

Under control of signal *winit* we select either the output of the initialization circuit (winit in the figure) or the output of the walk extension circuit (wext in the figure). In designs that follow this signal is provided by a corresponding control automaton, which actually controls the

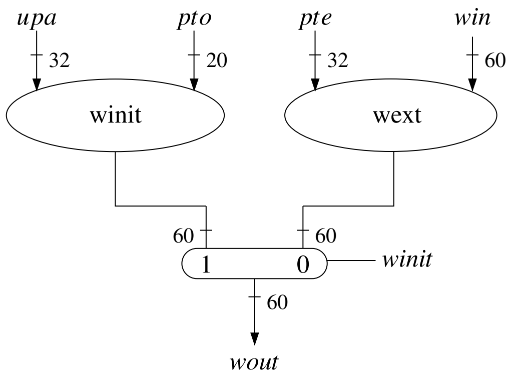


Fig. 22: Circuit for walk creation (initialization and extension, data paths)

walking unit. For the guest walking unit this signal is provided by the simple control automaton, whereas for the user walking unit — by the nested control automaton. For convenience, we also introduce signal *wext* which, as the name suggests, indicates that a walk extension is performed. The definitions are identical for both units walking: *winit* is raised whenever transition $(t1)$ is taken; *wext* — whenever the memory access finishes while the translation is not aborted.

$$\begin{aligned} winit(wunit_X) &\equiv idle_X \wedge (t1)_X \\ wext(wunit_X) &\equiv fetch\text{-}pte_X \wedge /mbusy \wedge treq_X \end{aligned}$$

Data Paths of Nested MMU

We manage to complete the construction of nested MMU with very little effort. All that is left to do are the connections of the two walking units with the corresponding walk registers and very few extra circuits, which we present next, and connections of the interfaces of components we use, which we present immediately after.

In case of the nested call, the right side of the nested MMU performs translation of the current base address of the user walk (w_U), whereas in case of a simple address translation, it translates the universal page address (*upa*) provided to the nested MMU as an external input.

$$wunit_G.upa = \begin{cases} uba(w_U) & nested\text{-}call \\ mmu.upa & \text{otherwise} \end{cases}$$

In the lines above we used the *universal base address* (*uba*) of a universal user walk.

$$w.upa \in A_U \;\rightarrow\; uba(w) = w.vm \circ 0_8 \circ w.ba$$

The latter address is fed into the guest walk register on nested translation calls. For intuition we refer to Fig. 23. Either way, the page table origin used by the guest walking unit is, obviously, taken directly from the nested MMU inputs.

$$wunit_G.pto = mmu.pto_G$$

The input of the guest walk register is connected to the outputs of the TLB component and the guest walking unit. The output of the TLB component is selected (by the multiplexer) once the TLB component contains the requested translation.

$$w_G.in = \begin{cases} tlb.wout & tlb.hit \\ wunit_G.wout & \text{otherwise} \end{cases}$$

This is done in order to flatten the differences between the ways in which translation of the $uba(w_U)$ is obtained, and thus to simplify the correctness proof in Chap. 5. In case the output

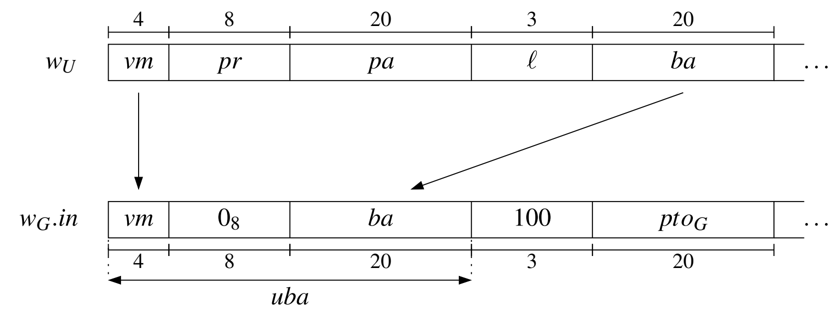


Fig. 23: Initialization of the guest walk register before a nested call

translation is obtained in the translation process, the control mechanism of the simple MMU guarantees that the guest walk register contains this translation. Otherwise, if the translation is output immediately and no prior translation process is done, the guest walk register contains some obsolete data (from the previous translations), unless the TLB output is written there.

For technical reasons we distinguish between the *special* and *standard* clocking of the guest walk register[7].

$$w_G.ce \equiv w_G.ce_{spc} \vee w_G.ce_{std}$$

As the naming suggests, the special clock enable signal handles the case in which the TLB output is clocked-in, whereas the standard clock enable signal, which is equivalent to the corresponding signal from [LOP], covers the remaining cases.

$$\begin{aligned} w_G.ce_{spc} &\equiv \mathit{nested\text{-}call} \wedge \mathit{tlb.hit} \\ w_G.ce_{std} &\equiv \mathit{winit}(\mathit{wunit}_G) \vee \mathit{wext}(\mathit{wunit}_G) \end{aligned}$$

The guest walk register is fed (back) into the walking unit

$$\mathit{wunit}_G.\mathit{win} = w_G$$

and into another small circuit (a multiplexer) which computes the address of the guest page table entry used to extend the current guest walk.

$$\mathit{ptea}_G = w_G.ba \circ \begin{cases} w_G.px_2 \circ 0_2 & w_G.\ell[2] \\ w_G.px_1 \circ 0_2 & \text{otherwise} \end{cases}$$

Also, the guest walk register is (indirectly) connected to the walk input of the TLB component s.t. the guest walk can be stored in the TLB cache. The details about connection to the TLB interface are considered below in this section.

We repeat the definitions above for connections of the user walking unit (on the LHS of the MMU component). They turn out to be considerably simpler and nearly self-explaining.

$$\begin{aligned} \mathit{wunit}_U.\mathit{upa} &= \mathit{mmu.upa} \\ \mathit{wunit}_U.\mathit{pto} &= \mathit{mmu.pto}_U \\ \mathit{wunit}_U.\mathit{win} &= w_U \end{aligned}$$

The universal page address and the page table origin are taken directly from the inputs of the nested MMU. Note that the user walking unit inputs the user page table origin (pto_U) instead of the one for guests (pto_G), which in turn is obvious. The walk input of the user walking unit is taken directly from the user walk register.

[7] Stepping of the TLB component will be associated with the clocking of the walk registers, same as in [LOP]. At the same time, updates of the guest walk register on the special clock enable signals should not produce any specification steps.

The user walk (w_U) register receives the data only from the user walking unit. The register is clocked on initialization of the user walk and whenever the memory access (user walk extension phase) finishes in presence of translation request.

$$\begin{aligned} w_U.in &= wunit_U.wout \\ w_U.ce &\equiv winit(wunit_U) \vee wext(wunit_U) \end{aligned}$$

The user page table entry address computed on the LHS of the MMU component depends also on the current guest walk (w_G). Formally, the circuit inputs the composition of two walks (user and guest) and outputs the address as specified below.

$$ptea_U = w_G.ba \circ \begin{cases} w_U.px_2 \circ 0_2 & w_U.\ell[2] \\ w_U.px_1 \circ 0_2 & \text{otherwise} \end{cases}$$

The memory is accessed at this address ($ptea_U$) and at the guest page table entry address ($ptea_G$) resp. in the control states *fetch-pte*$_U$ and *fetch-pte*$_G$. This implements the read of the user's and guest's page tables resp. Note that the accesses are coming from both sides of the MMU. Since the MMU component is connected only to one port of the memory system, with a simple separation of accesses we can define a single page table entry address from the other two as follows.

$$ptea = \begin{cases} ptea_U & \textit{fetch-pte}_U \\ ptea_G & \text{otherwise} \end{cases}$$

Then a single memory data output — page table entry (pte) — is available to both sides of the nested MMU and fed into the user and guest walking units.

$$pte_U = pte_G = pte$$

The memory interface does not change. We define it later in this section.

Finally, we specify outputs of the nested MMU component. Data output does not change and remains simply the data output of the hardware TLB.

$$mmu.wout = tlb.wout$$

But simplicity in the definition above is misleading. Recall, now the hardware TLB is used (alternately) by two different state machines, namely by the control automata of the simple and the nested MMU. In fact, both automata share the same output ($tlb.wout$) of the hardware TLB for their translation look-ups. For the MMU busy signal we rely on two busy signals coming from both sides of the nested MMU. Whenever at least one of these signals is raised the entire component becomes busy.

$$mmu.busy \equiv busy_U \vee busy_G$$

The busy signal coming from the user/guest side of the nested MMU is produced by the respective state machine as follows.

$$busy_X \equiv /idle_X \vee (t1)_X$$

Note, in case of a simple (not nested) translation, which keeps the nested control automaton in its idle state ($idle_U$), the nested MMU remains busy until the simple translation request is served.

Below we cover connections of interfaces. These connections are straightforward, so we explain them rather in a formal manner and give only brief explanations of the corresponding formulas.

Memory Interface

The nested MMU starts the memory access in the following two situations: i) when the current user walk (w_U) is being extended, and ii) when the current guest walk (w_G) is being extended.

$$mreq \equiv mreq_U \vee mreq_G$$

In the first case, the memory is accessed at the address $ptea_U$ by the nested control automaton, which resides in state $fetch\text{-}pte_U$, as we defined previously in this section. In the second case, the memory access is performed to the address $ptea_G$ and initiated by the simple control automaton, which resides in state $fetch\text{-}pte_G$ (same as in [LOP]). In both cases the memory is accessed at the address $ptea$, as we defined above in this section.

$$ma = ptea$$

Formally, we select the correct memory word using the third bit of the memory address and notation from Sect. 1.1.

$$pte = \begin{cases} mout_H & ma[2] \\ mout_L & \text{otherwise} \end{cases}$$

TLB Interface

Next we connect the hardware TLB, which we designed in Sect. 4.1 specifically to support the nested translation.[8] The interface of the hardware TLB was depicted in Fig. 14. Conveniently we split it into three parts, according to the form of performed access: i) translation look-up, ii) translation registering, and iii) translation invalidation. Connections to each of these parts are described below. Recall, at most one of these parts can be used at a time, i.e., the hardware TLB cannot handle simultaneous requests to different parts of its interface.

i) Requests for translation look-ups are coming from both sides of the nested MMU component. In both cases the request is initiated only when the corresponding control automaton resides in its $idle_U$ state.

$$tlb.trans \equiv idle_U \wedge treq_U \vee idle_G \wedge treq_G$$

In the first case, under the nested translation request, the hardware TLB is checked to contain a translation for input of the user walking unit (upa_U). In turn, in the second case, under the simple translation request, the hardware TLB is checked to contain a translation for input of the guest walking unit (upa_G).

$$tlb.upa = \begin{cases} upa_G & treq_G \\ upa_U & \text{otherwise} \end{cases}$$

ii) Writes are simple. The hardware TLB is written either i) in state $write\text{-}tlb_U$ of the nested control automaton or ii) in the counterpart control state $write\text{-}tlb_G$ of the simple control automaton. Note, writes are performed only in case the translation request was not aborted (see Sect. 4.3).

$$tlb.store \equiv write\text{-}tlb_U \wedge treq_U \vee write\text{-}tlb_G \wedge treq_G$$

In the first case, composition of the current user (w_U) and guest (w_G) walks is placed into the hardware TLB. In the second case, the hardware TLB is fed with the guest walk (w_G) as before.

[8] Due to certain limitations of the construction developed in [LOP] (namely, due to the absence of ASIDs and mechanisms to invalidate multiple translations at a time), there was a necessity to redesign the hardware TLB in order to make the nested translation possible.

$$tlb.win = \begin{cases} w_U \circ w_G & \textit{write-tlb}_U \\ w_G & \text{otherwise} \end{cases}$$

Additionally we (always) place into the hardware TLB the page address of the current guest walk (w_G). Details of the latter we covered in Sect. 4.1.1.

$$tlb.gin = w_G.pa$$

iii) Finally, we describe the invalidation part. Obviously, one only activates the invalidation signal of the hardware TLB if there is a corresponding processor request to the nested MMU component. Recall that our construction supports the following three invalidation mechanisms: i) *flush*, when all of the cached translations are dropped, ii) *vmflush*, when only translations for a certain (input) VM are dropped, and iii) *invlpg*, when only translations for a certain (input) universal page address are dropped.

$$tlb.inval \equiv flush \vee vmflush \vee invlpg$$

In hardware we distinguish between the cases above using the invalidation mask.

$$tlb.invm = \begin{cases} 111 & flush \\ 011 & vmflush \wedge /flush \\ 000 & \text{otherwise} \end{cases}$$

The address or VM to invalidate is passed through the invalidation address. Unless invalidation of the hardware TLB is requested, for technical reasons we always pass the universal page address of the input walk (*tlb.win*) as a "victim" address to meet the operating conditions of the hardware TLB (see Sect. 4.1.1).

$$tlb.inva = \begin{cases} mmu.inva & tlb.inval \\ tlb.win.upa & \text{otherwise} \end{cases}$$

Note, in case of *vmflush*, the ID of "victim" VM is passed through the upper portion of invalidation address, i.e., *tlb.inva.vm*.

Implementation Correctness

In the following simple lemma we justify our computation of the page table entry addresses used internally by i) the simple and ii) the nested MMU.

Lemma 20.

$$pteaG(h) = ptea(w_G) \quad (1)$$
$$pteaU(h) = ptea(w_U \circ w_G) \quad (2)$$

Proof of lemma 20.1. For the page table entry address of the simple MMU we show:

$$\begin{aligned} pteaG(h) &= w_G.ba \circ \begin{cases} w_G.px_2 \circ 00 & w_G.\ell[2] \\ w_G.px_1 \circ 00 & \text{otherwise} \end{cases} \\ &= \begin{cases} ptea(w_G.ba, w_G.px_2) & w_G.\ell[2] \\ ptea(w_G.ba, w_G.px_1) & \text{otherwise} \end{cases} \\ &= ptea(w_G.ba, w_G.px_{level(w_G)}) \\ &= ptea(w_G). \end{aligned}$$ □

In the proof lines below we use the shorthand

$$w_N \equiv w_U \circ w_G.$$

Proof of lemma 20.2. For the page table entry address of the nested MMU we argue similarly:

$$\begin{aligned}
ptea_U(h) &= w_G.ba \circ \begin{cases} w_U.px_2 \circ 00 & w_U.\ell[2] \\ w_U.px_1 \circ 00 & \text{otherwise} \end{cases} \\
&= (w_U \circ w_G).ba \circ \begin{cases} (w_U \circ w_G).px_2 \circ 00 & (w_U \circ w_G).\ell[2] \\ (w_U \circ w_G).px_1 \circ 00 & \text{otherwise} \end{cases} \\
&= \begin{cases} ptea(w_N.ba, w_N.px_2) & w_U.\ell[2] \\ ptea(w_N.ba, w_N.px_1) & \text{otherwise} \end{cases} \\
&= ptea(w_N.ba, w_N.px_{level(w_N)}) \\
&= ptea(w_U \circ w_G). \qquad \square
\end{aligned}$$

4.3 Liveness

In this section we show that the implementation is live, i.e., that translation requests are eventually served by the nested MMU. In order to develop some intuition, first we consider a typical use case: translation of a user address. Starting from state $idle_U$, we go through the remaining states of the nested MMU and informally argue about the control mechanism. Note, we assume the absence of *reset*.

i) In case of the nested translation request ($treq_U$), the control automaton of the nested MMU leaves its idle state on transition ($t1$) if the request cannot be served immediately from the TLB cache ($/tlb.hit$); control goes to state *nested-call*, while the user walk register (w_U) is initialized with the requested *upa* as the page address and the user PTO (pto_U) as the base address.
ii) In state *nested-call* a translation request to the *simple CA* is raised. The request is for translation of the base address of the user walk register (pto_U). Liveness of the *simple CA* is assumed at this stage.
iii) When serving of the simple request finishes, control is either returns to state *fetch-pte*$_U$ or exists into state *write-tlb*$_U$. Transition ($t8$) to state *write-tlb*$_U$ is taken in case of a fault of the nested translation (see Sect. 3.3.3) or simply when translation of the user address (upa) successfully finishes.
iv) In state *fetch-pte*$_U$ the memory access is performed with the aim of extension of composed walk $w_U \circ w_G$ (see Sect. 3.2.2). Liveness of the memory system is assumed at this stage.
v) Once the walk extension in state *fetch-pte*$_U$ is finished, we check the faultiness of the extended walk ($wext_U$) and update the user walk register (w_U). In case the extension was faulty ($f(wext_U)$), control exists into the *write-tlb*$_U$ state. Otherwise, a new round of translation starts in state *nested-call*.
vi) Last, in state *write-tlb*$_U$ the translation process finishes and control returns to state $idle_U$.

In order to show things formally we require somewhat more technical definitions. First of all, to shorten the forthcoming proofs, we abbreviate functions Z of hardware configuration h in cycle t as

$$Z(t) = Z(h^t).$$

Hardware cycles t in which the process of address translation starts resp. ends are identified by the following predicates.

$$\begin{aligned}
t\text{-}start(t) &\equiv idle(t) \wedge /idle(t+1) \\
t\text{-}end(t) &\equiv idle(t) \wedge /idle(t-1)
\end{aligned}$$

Analogously we identify hardware cycles t in which the process of address translations terminates due to the loss of translation request. In the such cases we say that translation process was *aborted*.

$$t\text{-}abort(t) \equiv treq(t-1) \wedge busy(t-1) \wedge /treq(t)$$

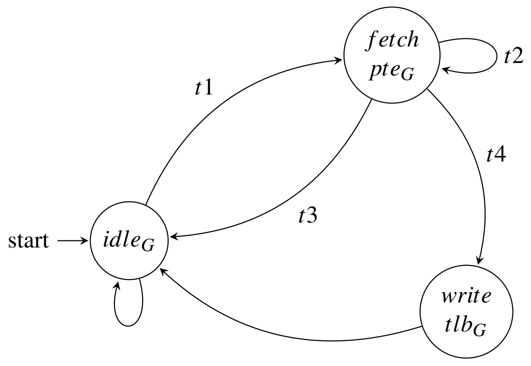


Fig. 24: Control automaton of the simple MMU

Next we introduce notation for intervals of cycles in which the translation request is continuous. Also we abbreviate the intervals in which the first and the last cycles are resp. the starting and the ending cycles of the address translation process.

$$treq[t_1:t_2] \equiv (t_1 < t_2) \wedge \forall t \in [t_1:t_2] : treq(t)$$
$$treg[t_1:t_2] \equiv treq[t_1:t_2] \wedge t\text{-}start(t_1) \wedge t\text{-}end(t_2)$$

The latter intervals of cycles we call *regular* translation phases. Below we consider liveness of two control automata (simple and nested), which both operate on a single hardware TLB. Throughout the proofs one can easily see that the following invariants are maintained by the hardware.

Invariant 3.

$$w \in tlbset(t) \;\rightarrow\; f(w) \vee w.\ell[0] \quad (1)$$
$$f(w_X) \vee w_X.\ell[0] \;\rightarrow\; /fetch\text{-}pte_U \wedge /fetch\text{-}pte_X \quad (2)$$
$$f(w_U, w_G) \vee (w_U \overset{a}{\not\equiv} w_G) \;\rightarrow\; /fetch\text{-}pte_U \quad (3)$$

4.3.1 Simple Translations

The simple control automaton is depicted in Fig. 20. In addition to labels introduced in the figure, using label $(t4)$ we refer to a transition between states *fetch-pte*$_G$ and *write-tlb*$_G$. For convenience in Fig. 24 we give a construction of the automaton with this additional label. Transitions of the simple control automaton are listed below.

$$(t1) \equiv /reset \wedge treq_G \wedge /tlb.hit$$
$$(t2) \equiv /reset \wedge mbusy \;\equiv\; /(t3) \wedge /(t4)$$
$$(t3) \equiv reset \vee /mbusy \wedge /treq_G$$
$$(t4) \equiv /reset \wedge /mbusy \wedge treq_G \wedge (f(wext_G) \vee wext_G.\ell[0])$$

Several restrictions on input signals are necessary for the simple MMU to function properly. We formulate these restrictions in the form of operating conditions; these conditions are collected below.

$$busy_G(t) \rightarrow (treq_G[t_0:t] \rightarrow upa_G(t) = upa_G(t_0)) \quad (1)$$
$$treq_G(t) \rightarrow /tlb.ireq(t) \quad (2)$$

Properties of Simple CA

In the following two lemmas we show some important properties of the simple control automaton. Essentially, these lemmas are the counterparts of the corresponding lemmas from [LOP] about properties of the local control automaton.

Lemma 21.

$$\mathit{fetch\text{-}pte}_G(t) \;\rightarrow\; \exists t' > t : /\mathit{fetch\text{-}pte}_G(t') \;\vee\; \mathit{level}(w_G(t')) < \mathit{level}(w_G(t))$$

Proof of lemma 21. There are three transitions possible in state $\mathit{fetch\text{-}pte}_G$ in cycle t: $(t2)$, $(t3)$, and $(t4)$. In case the latter two transitions are taken, the control goes either to state idle_G or $\mathit{write\text{-}tlb}_G$. In both cases we trivially obtain for $t' = t+1$:

$$/\mathit{fetch\text{-}pte}_G(t').$$

If transition $(t2)$ is taken, we deduce, obviously, $/\mathit{reset}(t)$, and either

$$\mathit{mbusy}(t) \qquad \text{i)}$$

or

$$/\mathit{mbusy}(t) \wedge \mathit{treq}_G(t) \wedge /(f(\mathit{wext}_G(t)) \vee \mathit{wext}_G.\ell[0]). \qquad \text{ii)}$$

In the first case we use liveness of the memory system to argue that

$$\exists t'' > t : /\mathit{mbusy}(t'')$$

which brings us into the second case. There, we argue that

$$\mathit{fetch\text{-}pte}_G(t) \wedge /\mathit{mbusy}(t) \;\rightarrow\; w_G(t+1) = \mathit{wext}_G(t)$$

from the definitions of the update enable and input signals of the guest walk register. Using correctness of the circuit which implements walk extension, we conclude for $t' = t+1$:

$$\mathit{level}(w_G(t')) < \mathit{level}(w_G(t)). \qquad \square$$

Lemma 22. *Assume the absence of reset in cycles* $[t_0 : t_1]$. *Let*

$$\mathit{fetch\text{-}pte}_G(t_0) \;\wedge\; t_1 = \min\{t' > t_0 \mid /\mathit{fetch\text{-}pte}_G(t')\} \;\wedge\; \mathit{treq}_G[t_0 : t_1].$$

Then

$$f(w_G(t_1)) \;\vee\; w_G(t_1).\ell[0] \qquad \text{i)}$$

and

$$w_G(t_1).\mathit{upa} = \mathit{upa}_G(t_0). \qquad \text{ii)}$$

Proof of lemma 22. From the absence of *reset* and stable translation request we conclude that

$$\forall t \in [t_0 : t_1] : /(t3)(t)$$

and therefore

$$\mathit{write\text{-}tlb}_G(t_1).$$

Also we argue that transition $(t4)$ was taken in cycle $(t_1 - 1)$. This immediately gives i), since

$$w_G(t_1) = \mathit{wext}_G(t_1 - 1)$$

from the definitions of the update enable and input signals of the guest walk register. For ii) we refer to operating condition 1 and argue analyzing data paths of the MMU that

$$w_G(t)\mathit{in}.\mathit{upa} = \begin{cases} \mathit{upa}_G(t) & \mathit{winit}_G(t) \\ w_G(t).\mathit{upa} & \text{otherwise} \end{cases}$$

given that circuits for walk initialization and extension are implemented correctly. $\square$

Liveness of Simple CA

In the remainder of this section we elaborate on liveness of the simple control automaton and results of the simple address translation. Thus, in the following lemma we show that every started *simple* translation eventually ends.

Lemma 23.

$$\textit{t-start}_G(t) \;\rightarrow\; \exists t' > t:\; \textit{t-end}_G(t')$$

Proof of lemma 23. We cover all cases that possibly occur in the process of simple translation. In every subsequent case we consider a complement to everything encountered so far. Obviously, every translation process starts in the idle state of the control automaton:

$$\textit{idle}_G(t).$$

- transition $(t1)$ is taken in cycle t. The control goes to state *fetch-pte*$_G$.

  $$\textit{fetch-pte}_G(t+1)$$

  Using lemma 21 we conclude for some cycle $t_1 > t+1$

  $$/\textit{fetch-pte}_G(t_1) \qquad \text{i)}$$

  or

  $$\textit{fetch-pte}_G(t_1) \wedge \textit{level}(w_G(t_1)) < \textit{level}(w_G(t+1)). \qquad \text{ii)}$$

  In the first case the control goes either to state *idle*$_G$ (abort) or to state *write-tlb*$_G$.

  $$\begin{aligned} \textit{idle}_G(t_1) &\rightarrow t' = t_1 \\ \textit{write-tlb}_G(t_1) &\rightarrow t' = t_1 + 1 \end{aligned}$$

- in the second case the translation continues in state *fetch-pte*$_G$, but the extended walk decreases its level. Moreover, this time the extended walk has level one, therefore following the same argument as above we conclude for some cycle $t_2 > t_1$

  $$/\textit{fetch-pte}_G(t_2).$$

  Again, translation is either aborted or not, resp. the control goes either to state *idle*$_G$ or to state *write-tlb*$_G$.

  $$\begin{aligned} \textit{idle}_G(t_2) &\rightarrow t' = t_2 \\ \textit{write-tlb}_G(t_2) &\rightarrow t' = t_2 + 1 \end{aligned} \qquad \square$$

In the next lemma we argue that every regular *simple* translation results in a TLB hit. As we argue later, the latter hit signals that the hardware TLB contains a translation for the universal (guest) address which was requested to translate.

Lemma 24.

$$\textit{treg}_G[t_0 : t_1] \;\rightarrow\; \textit{tlb.hit}(t_1)$$

Proof of lemma 24. Given that translation is regular, we immediately obtain

$$\forall t \in [t_0 : t_1] : /\textit{t-abort}_G(t).$$

Next we argue along the lines of the proof above. We consider only those cases in which translation is not aborted.

- translation is served directly from the TLB. For $t_0 = t_1$ we conclude

  $$\textit{tlb.hit}(t_0)$$

  immediately from the definition of transition $(t1)$ (which is not taken).

- transition $(t1)$ is taken in cycle t_0. The control goes to state *fetch-pte*$_G$

  $$\textit{fetch-pte}_G(t_0+1)$$

  and stays there (no abort) until some cycle $t' > t_0+1$ (lemma 21). For cycles until t' there is nothing to show:

  $$\forall t \in [t_0 : t'] : /\textit{t-end}_G(t).$$

  Again, since translation is not aborted, the control goes to state *write-tlb*$_G$.

  $$\textit{write-tlb}_G(t')$$

  There the content of the walk register is written into the TLB. From the TLB hardware specification we get

  $$\textit{tlbset}(t'+1) \cap \{w \mid w.\textit{upa} = w_G(t').\textit{upa}\} = \{w_G(t')\}.$$

  Using lemma 22 for cycle $t' > t_0+1$ and operating condition 1 we obtain

  $$w_G(t').\textit{upa} = \textit{upa}_G(t_0+1) = \textit{upa}_G(t_1).$$

  When control returns to the idle state in cycle $t'+1 = t_1$, we conclude

  $$\textit{tlb.hit}(t'+1)$$

  since due to operating condition 2 we have

  $$/\textit{tlb.ireq}(t'). \qquad \square$$

From the latter lemma we trivially conclude (from the MMU interconnect):

$$\textit{treg}[t_0 : t] \;\rightarrow\; \textit{wout}(t) \in \textit{tlbset}(t)$$

and

$$\textit{wout}(t).\textit{upa} = \textit{upa}_G(t).$$

And from part (1) of invariant 3 we conclude:

$$f(\textit{wout}(t)) \;\vee\; \textit{wout}(t).\ell[0].$$

In the last lemma proven in this section we show that the simple control automaton always eventually returns to its idle state.

Lemma 25.

$$/\textit{idle}_G(t) \;\rightarrow\; \exists t' > t : \; \textit{idle}_G(t')$$

Proof of lemma 25. By induction on number t of the hardware cycles. For the base case ($t = 0$) there is nothing to show since immediately after reset the control resides in its idle state.

$$\textit{idle}_G(0)$$

For the induction step from t to $t+1$ we argue as follows. Clearly, we cover only

$$/\textit{idle}_G(t+1)$$

since otherwise, as usual, there is nothing to show. Next, we split cases on whether the control resided in the idle state in cycle t or not.

- $\textit{idle}_G(t) = 1$. Directly from the definitions we have

  $$\textit{t-start}_G(t).$$

  Therefore, applying lemma 23 proves the claim for some cycle $t' > t$.
- $\textit{idle}_G(t) = 0$. In this case we argue using the induction hypothesis, which for some cycle $t' > t$ gives

  $$\textit{idle}_G(t').$$

  Since the control does not reside in its idle state in cycle $t+1$ (as we assumed), for cycle t' we have

  $$t' > t+1$$

  which completes the induction step, and therefore the proof. $\square$

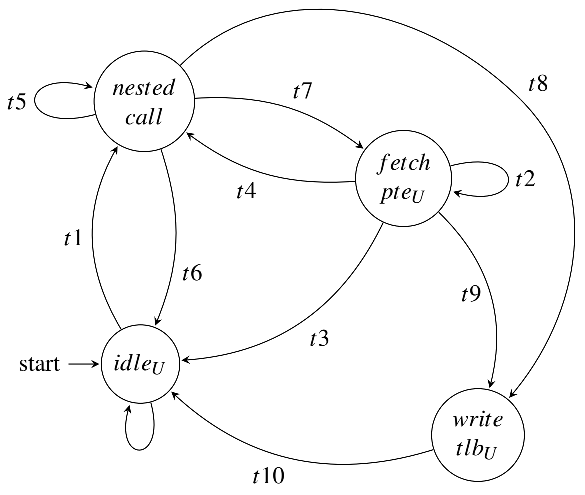


Fig. 25: Control automaton of the nested MMU

4.3.2 Nested Translations

The nested control automaton is depicted in Fig. 21. In addition to labels introduced in the figure, using label $(t9)$ we refer to a transition between states *fetch-pte$_U$* and *write-tlb$_U$*, whereas using label $(t10)$ — to a transition between states *write-tlb$_U$* and *idle$_U$*. For convenience in Fig. 25 we give a construction of the automaton with additional labels. Transitions of the nested control automaton are listed below. In addition to transition

$$(t1) \equiv /reset \wedge treq_U \wedge /tlb.hit$$

which takes control out of the idle state, we have four transitions emanating from state *fetch-pte$_U$*

$$\begin{aligned}(t2) &\equiv /reset \wedge mbusy\\ (t3) &\equiv reset \vee /mbusy \wedge /treq_U\\ (t4) &\equiv /(t2) \wedge /(t3) \wedge /f(wext_U)\\ (t9) &\equiv /(t2) \wedge /(t3) \wedge f(wext_U)\end{aligned}$$

as well as another four transitions from state *nested-call.*

$$\begin{aligned}(t5) &\equiv /reset \wedge busy_G\\ (t6) &\equiv reset \vee /busy_G \wedge /treq_U\\ (t7) &\equiv /(t5) \wedge /(t6) \wedge /(f(w_U, tlb.wout) \vee w_U.\ell[0])\\ (t8) &\equiv /(t5) \wedge /(t6) \wedge (f(w_U, tlb.wout) \vee w_U.\ell[0])\end{aligned}$$

The operating conditions remains the same as for the simple control automaton. For convenience we reformulate these conditions to match the interface of the nested MMU.

$$busy(t) \rightarrow (treq[t_0 : t] \rightarrow upa(t) = upa(t_0)) \tag{1}$$
$$treq(t) \rightarrow /tlb.ireq(t) \tag{2}$$

Properties of Nested CA

The first two lemmas directly follow from liveness of the cache memory system (lemma 26) and simple control automaton (lemma 27).

Lemma 26.

$$\textit{fetch-pte}_U(t) \;\rightarrow\; \exists t' > t : /\textit{fetch-pte}_U(t')$$

Lemma 27.

$$\textit{nested-call}(t) \;\rightarrow\; \exists t' > t : /\textit{nested-call}(t')$$

In the next lemma we capture the following property: whenever the nested control automaton leaves states *fetch-pte*$_U$ or *nested-call* with a persistent request for nested translation, the user walk register contains a translation for the universal (user) address which was requested to translate; moreover, the latter translation is incomplete only if composition of the walks in the walk registers is faulty.

Lemma 28. *Assume the absence of reset in cycles* $[t_0 : t_1]$. *Let*

$$(\textit{fetch-pte}_U \vee \textit{nested-call})(t_0) \;\wedge\; t_1 = \min\{t' > t_0 \mid \textit{write-tlb}_U(t')\} \;\wedge\; \textit{treq}_U[t_0 : t_1].$$

Then

$$f(w_U(t_1) \circ w_G(t_1)) \;\vee\; w_U(t_1).\ell[0] \qquad \text{i)}$$

and

$$w_U(t_1).upa = upa_U(t_0). \qquad \text{ii)}$$

Proof of lemma 28. From the absence of *reset* and stable translation request we conclude that

$$\forall t \in [t_0 : t_1] : /(t3)(t) \wedge /(t6)(t).$$

State *write-tlb*$_U$ can only be reached through transitions $(t8)$ and $(t9)$. In the first case we argue

$$\begin{aligned}(t8)(t_1-1) &\rightarrow f(w_U(t_1-1), \textit{tlb.wout}(t_1-1)) \;\vee\; w_U(t_1-1).\ell[0] \\ &\rightarrow f(w_U(t_1), w_G(t_1)) \;\vee\; w_U(t_1).\ell[0]\end{aligned}$$

from the definitions of the update enable and input signals of the walk registers. The last line implies i) by definition of the walk composition. In the second case we deduce

$$\begin{aligned}(t9)(t_1-1) &\rightarrow f(\textit{wext}_U(t_1-1)) \\ &\rightarrow f(w_U(t_1))\end{aligned}$$

which again implies i) by definition of the walk composition. For part ii) we argue in exactly the same way as in the proof of lemma 22. □

Liveness of Nested CA

The following three lemmas are counterparts of the corresponding lemmas for the simple control automaton. The structure of their proofs will therefore repeat. Thus, in the first lemma below we show that every translation eventually ends.

Lemma 29.

$$\textit{t-start}(t) \;\rightarrow\; \exists t' > t :\; \textit{t-end}(t')$$

Proof of lemma 29. We consider only nested requests

$$\textit{treq}_U(t)$$

since otherwise the control stays in the idle state while the simple control automaton is used. Liveness of the latter automaton was already proven (see lemma 23 above). Every nested translation starts in the idle state of the nested control automaton:

$$\textit{idle}_U(t).$$

- transition $(t1)$ is taken in cycle t. The control goes to state *nested-call*.

$$\mathit{nested\text{-}call}(t+1)$$

From lemma 27 we know there is a cycle $t_1 > t+1$ such that

$$/\mathit{nested\text{-}call}(t_1).$$

In case the control returns to the idle state (via transition $(t6)$) or goes to state *write-tlb*$_U$ (via transition $(t8)$) we conclude trivially:

$$\begin{aligned} \mathit{idle}_U(t_1) &\rightarrow t' = t_1 \\ \mathit{write\text{-}tlb}_U(t_1) &\rightarrow t' = t_1 + 1. \end{aligned}$$

- otherwise, the control goes to state *fetch-pte*$_U$ (via transition $(t7)$)

$$\mathit{fetch\text{-}pte}_U(t_1)$$

in which it stays until some cycle $t_2 > t_1$ (lemma 26). Again we consider two cases: either i) the control returns to the idle state (via transition $(t3)$) or goes to state *write-tlb*$_U$ (via transition $(t9)$), or ii) otherwise, the control returns to state *nested-call* (via transition $(t4)$). In the first case we obtain trivially:

$$\begin{aligned} \mathit{idle}_U(t_2) &\rightarrow t' = t_2 \\ \mathit{write\text{-}tlb}_U(t_2) &\rightarrow t' = t_2 + 1. \end{aligned}$$

- in the second case

$$\mathit{nested\text{-}call}(t_2)$$

we additionally argue that the level of the user walk decreases

$$\begin{aligned} \mathit{level}(w_U(t_2)) &< \mathit{level}(\mathit{wext}_U(t_2 - 1)) \\ &= \mathit{level}(w_U(t_1)) \end{aligned}$$

relying on correctness of circuits for the walk extension and using definitions of the update enable and input signals of the user walk register. This brings us into a loop (between the control states *nested-call* and *fetch-pte*$_U$), which cannot last forever. Since the user walk is initialized to have the second level[9]

$$\mathit{winit}_U(t) \;\rightarrow\; \mathit{level}(w_U(t+1)) = 2$$

we conclude that at some point transition $(t7)$ is no longer available and the loop is broken.

□

In the next lemma we argue that every regular translation results in a TLB hit. The latter hit signal, as we argue afterwards, indicates that a translation for the universal address which was requested to translate is present in the hardware TLB.

Lemma 30.

$$\mathit{treg}[t_0 : t_1] \;\rightarrow\; \mathit{tlb.hit}(t_1)$$

[9] We consider the page tables of depth 2 here, but that is completely technical parameter. In general, for page tables of depth d and *nestedness* of the privilege levels n the total number of walk extensions necessary to translate the virtual address of the highest privilege level is

$$(d+1)^{n-1} - 1.$$

Proof of lemma 30. Again we argue only about the nested requests

$$treq_U(t)$$

since otherwise the control stays in the idle state while the simple control automaton is used. Given that translation is regular, we immediately obtain

$$\forall t \in [t_0 : t_1] : /t\text{-}abort_U(t).$$

Thus, we consider only those cases in which translation is not aborted.

- translation is served directly from the TLB. We conclude for $t_0 = t_1$

  $$tlb.hit(t_0)$$

  immediately from the definition of transition $(t1)$ (which is not taken).
- transition $(t1)$ is taken in cycle t_0. The control goes to state *nested-call.*

  $$nested\text{-}call(t_0 + 1)$$

  Given that translation is regular in cycles $[t_0 : t_1]$ (no abort), we conclude using liveness of the nested control automaton (lemma 29) that the translation process ends with transition $(t10)$ — between states $write\text{-}tlb_U$ and $idle_U$. Therefore in cycle $t' = t_1 - 1$ the hardware TLB is written

  $$write\text{-}tlb_U(t')$$

  with the composition of walks in the walk registers such that in cycle $t' + 1 = t_1$ we have

  $$tlbset(t' + 1) \cap \{w \mid w.upa = w_U(t').upa\} = \{w_U(t') \circ w_G(t')\}.$$

  Using lemma 28 for cycle $t' > t_0 + 1$ and operating condition 1 we obtain

  $$w_U(t').upa = upa_U(t_0 + 1) = upa_U(t_1).$$

  When control returns to the idle state, we conclude

  $$tlb.hit(t' + 1)$$

  since due to operating condition 2 we have

  $$/tlb.ireq(t'). \qquad \square$$

From the latter lemma we trivially conclude (from the MMU interconnect):

$$treg[t_0 : t] \;\rightarrow\; wout(t) \in tlbset(t)$$

and

$$wout(t).upa = upa(t).$$

And from part (1) of invariant 3 we conclude:

$$f(wout(t)) \;\vee\; wout(t).\ell[0].$$

Finally, we argue that the nested control automaton always eventually returns to its idle state s.t. both control automata are found in the idle states. As a result, the nested MMU always eventually lowers the busy signal.

Lemma 31.

$$/idle(t) \;\rightarrow\; \exists t' > t : \; idle(t')$$

Proof of the latter lemma is analogous to proof of the corresponding lemma for the simple control automaton above (lemma 25), and therefore is omitted. This liveness result completes the implementation of the nested MMU. Correctness of the latter implementation is proven in the next chapter.

5

Correctness of NAT Implementation

In the previous chapter we introduced a hardware construction which implements the process of nested address translation. We continue referring to that construction using the name *nested MMU*, though sometimes we call it simply *MMU* for short. Still, when we refer to the original MMU design from [LOP], we explicitly use a name *original MMU* or *simple MMU*.

Construction of the nested MMU followed certain hardware specifications, which preceded every of its subcomponents. In this chapter we justify those hardware specifications in the usual manner — via correctness proofs.

5.1 Accessing MMU

In this chapter we are going to show that our implementation of the nested MMU works correctly in *all possible* hardware computations. For that purpose we formalize all accesses to the nested MMU in the form of *queries*. Under the queries we understand the data coming to the component's interfaces (input) while any of the request signals is high. Analyzing interfaces of the nested MMU from Sect. 4.2.1 we define query

$$qr \in K_{query}$$

to have the following components:

- $qr.upa \in \mathbb{B}^{32}$ — universal page address to provide/invalidate translation(s) for,
- $qr.ireq \in \mathbb{B}^{3}$ — invalidation requests; bits $[2:0]$ represent resp. requests for invalidation of all translations, translations of a virtual machine, as well as individual translations, and
- $qr.treq \in \mathbb{B}$ — translation request; output a translation for the requested address; in case a translation is not contained in the TLB, create it first.

We call a query *well-formed* if at most one of its request signals is raised.

$$wf(qr) \;\equiv\; (qr.treq + qr.ireq[0] + qr.ireq[1] + qr.ireq[2] \leq 1)$$

In case none of the request signals is raised, we say that the query is *void*.

$$void(qr) \;\equiv\; (qr.treq + qr.ireq[0] + qr.ireq[1] + qr.ireq[2] = 0)$$

According to the request signal raised we distinguish between translation and invalidation queries. Moreover, depending on the requested address we distinguish between simple and nested translation queries. Translation queries utilize some additional components:

- $qr.pto_G \in \mathbb{B}^{20}$ — guest page table origin,
- $qr.pto_U \in \mathbb{B}^{20}$ — user page table origin,
- $qr.data \in \mathbb{B}^{64}$ — data (memory line) to use in case of a walk extension, and
- $qr.drdy \in \mathbb{B}$ — control bit signaling that data provided are ready to use.

Application of queries is done simply by connecting their components to interfaces of the nested MMU. Namely, in order to apply query *qr* to nested MMU configuration *mmu* we connect the data inputs as well as the control inputs as follows. For the data inputs we have

$$\begin{aligned} mmu.upa &= qr.upa \\ mmu.pto_G &= qr.pto_G \\ mmu.pto_U &= qr.pto_U \\ mmu.inva &= qr.upa \\ mmu.mout &= qr.data \end{aligned}$$

whereas for the control inputs we have

$$\begin{aligned} mmu.treq &= qr.treq \\ mmu.invlpg &= qr.ireq[0] \\ mmu.vmflush &= qr.ireq[1] \\ mmu.flush &= qr.ireq[2] \\ mmu.mbusy &= \overline{qr.drdy}. \end{aligned}$$

The resulting configuration *mmu*′ is given by the hardware transition function the nested MMU.

$$mmu' = \delta_{mmu}(mmu, qr)$$

Formal definition of the latter function is omitted, however it can be easily extracted from the formal hardware specification of the nested MMU presented in Sect. 4.2.1.

Naturally, we introduce *query sequences*

$$qr : \mathbb{N} \to K_{query}$$

to be ordinary (infinite) sequences of queries. Note, we use the same notation to denote queries and query sequences. Hopefully, the difference will be clear from the context. We call a query sequence well-formed simply if every query in that sequence is well-formed.

$$wf(qr) \equiv \forall t : wf(qr[t])$$

Let *mmu∅* be the nested MMU hardware configuration after reset. Application of first *t* queries from query sequence *qr* naturally brings us to configuration

$$\Delta^t_{mmu}(mmu\emptyset, qr) = \begin{cases} \delta_{mmu}(\Delta^{t-1}_{mmu}(mmu\emptyset, qr), qr[t-1]) & t > 0 \\ mmu\emptyset & \text{otherwise.} \end{cases}$$

For convenience we abbreviate configuration above as

$$mmu^t_{qr} \equiv \Delta^t_{mmu}(mmu\emptyset, qr).$$

Clearly, we prove correctness of the nested MMU implementation not for all possible hardware computations, but for all computations in which the operating conditions of the nested MMU are respected. These conditions were formulated in Sect. 4.3, where we covered circuit correctness and liveness of the nested MMU. Here we reformulate the latter conditions to restrict the query sequences that we consider in the proofs below. Thus, we call a query sequence *valid* if i) the sequence is well-formed and ii) application of the sequence does not violate operating conditions of the nested MMU.

$$valid(qr) \equiv wf(qr) \wedge (mmu^t_{qr}.busy \wedge qr[t_0 : t].treq \to qr[t].upa = qr[t_0].upa)$$

5.2 Correctness Statement

In the end of this section (in Sect. 5.2.3) we state a (simulation) theorem claiming that the constructions made in Chap. 4 implement their specifications. In a nutshell this theorem claims that for every hardware cycle there is a configuration from the *general computation* (see Sect. 3.4) s.t. the *simulation relation* holds between the hardware TLB and the TLB component of that configuration.

Of course, in oder to formulate the latter theorem, we must provide the missing definitions. Thus, in Sect. 5.2.1 we specify how to construct the general computation from any valid MMU query sequence. The aforementioned simulation relation is defined in Sect. 5.2.2.

5.2.1 Stepping of TLB

In the original MMU design, stepping of the TLB component was associated with clocking of the walk register [LOP]. For the nested MMU component this concept does not change. We naturally transfer it to a more general MMU construction, which uses two hardware walk registers (guest and user) instead of one. Clocking these registers will "generate" different specification steps, though the difference is purely technical. The guest walk register generates steps of initialization or extension of the guest walks (identical to the original stepping), whereas the user walk register — steps of initialization or extension of the user walks (new). We differentiate between the generated steps using the following hardware signals.

$$\begin{aligned} tadd_G(mmu) &\equiv w_G.ce_{std} \\ tadd_U(mmu) &\equiv w_U.ce \end{aligned}$$

Whenever the guest walk register (w_G) is updated on the standard clock enable signal, it generates a *guest TLB step* ($tadd_G$). Every time the user walk register (w_U) is updated it generates a *user TLB step* ($tadd_U$). Then the total number of TLB steps generated in configuration h is obviously given by

$$tadd(mmu) = tadd_G(mmu) + tadd_U(mmu).$$

From simple analysis of the control logic of the nested MMU (Sect. 4.2.2) we argue that at most one TLB step can be generated in any given MMU configuration. Another trivial lemma follows.

Lemma 32.

$$tadd(mmu) \leq 1$$

Proof of lemma 32. Using properties of the nested control automaton we argue:

$$\begin{aligned} w_U.ce &\rightarrow treq_U \wedge (idle_U \vee fetch\text{-}pte_U) \\ &\rightarrow /sreq \wedge /nested\text{-}call. \end{aligned}$$

The last line implies the translation request signal of the simple control automaton is low (see Sect. 4.2.2). From there we immediately conclude:

$$/treq_G \;\rightarrow\; /w_G.ce_{std}. \qquad \square$$

For convenience we introduce few additional abbreviations. Similarly to the above, we extract the following two signals from the control logic of the nested MMU.

$$\begin{aligned} winit_X(mmu) &\equiv winit(mmu.wunit_X) \\ wext_X(mmu) &\equiv wext(mmu.wunit_X) \end{aligned}$$

We distinguish initialization cycles of the guest walk register from the corresponding cycles for the user walk register. For this purpose we use the predicates above ($winit_G$ and $winit_U$ resp. for the guest and the user walk register).

$$winit(mmu) = winit_G(mmu) + winit_U(mmu)$$

Since the walk registers are updated in the initialization cycles (Sect. 4.2.2), we immediately conclude from lemma 32 that at most one of these registers can be initialized in the given MMU configuration.

Lemma 33.

$$winit(mmu) \leq 1$$

In contrast to the overall specification, where invalidation of the TLB content occurs on passive TLB transitions within the processor core steps, the general specification defines the dedicated TLB steps to drop the stored translations (see Sect. 3.4.1).

$$tdrop(mmu) \equiv mmu.tlb.inval$$

Taking the newly introduced steps into account we define the total number of steps generated by the nested MMU in configuration *mmu* as

$$ns(mmu) = tadd(mmu) + tdrop(mmu).$$

Given that translation and invalidation queries are never served in parallel, using lemma 32 we obtain that at most one TLB step (out of three) can be generated in any given MMU configuration.

Lemma 34.

$$ns(mmu) \leq 1$$

Finally, using all the abbreviations above, we can specify the input passed to the specification machine (value of the stepping function).

$$\begin{aligned}
tadd_G(mmu) &\rightarrow s(mmu) = \begin{cases} (winit, upa_G, pto_G) & winit_G(mmu) \\ (wext, w_G, pte_G) & \text{otherwise} \end{cases} \\
tadd_U(mmu) &\rightarrow s(mmu) = \begin{cases} (winit, upa_U, pto_U) & winit_U(mmu) \\ (wext, w_U, pte_U) & \text{otherwise} \end{cases} \\
tdrop(mmu) &\rightarrow s(mmu) = \begin{cases} (drop, inva) & mmu.invlpg \\ (drop, inva.vm) & mmu.vmflush \\ (drop, all) & \text{otherwise} \end{cases}
\end{aligned}$$

Here we denote this particular input (for TLB steps) by $s(mmu)$, however below, in Sect. 5.2.1, we formalize it for a more general case of the non-sequential machines. There it becomes a component of a vector with the length $ns(h)$ — the number of all ISA steps performed in configuration h.

5.2.2 Simulation of TLB

First of all we introduce a simulation relation

$$sim_{tlb}(mmu, c)$$

which expresses that the TLB of (hardware) MMU configuration *mmu* is correctly simulated by the TLB of (software) ISA configuration c. Conveniently we split this simulation relation into two parts: sim_T and sim_W, for simulation of the TLB content and the walk registers resp.

$$sim_{tlb}(mmu, c) \equiv sim_T(mmu, c) \wedge sim_W(mmu, c)$$

Next we specify each part separately.

Simulation of TLB Content

For convenience we split the set of walks stored in the hardware TLB (*tlbset*) into the sets of hardware guest and user walks. We denote these sets by tlb_G and tlb_U resp. and define them as follows.

$$\begin{aligned} tlb_G(mmu) &= \{w \in tlbset(mmu.tlb) \mid w.upa \in A_G\} \\ tlb_U(mmu) &= \{w \in tlbset(mmu.tlb) \mid w.upa \in A_U\} \end{aligned}$$

Now the simulation of the hardware TLB content can be easily expressed: from the hardware guest walks we require to be contained in the software TLB, whereas from the hardware user walks — to be contained in the composed TLB.

$$\begin{aligned} sim_T(mmu,c) \;\equiv\; & tlb_G(mmu) \subseteq c.tlb \;\wedge \\ & tlb_U(mmu) \subseteq c.tlb^\circ \end{aligned}$$

Simulation of Walk Registers

Two hardware walk registers involved in the process of nested address translation — guest w_G and user w_U walk registers — were added for purely technical reasons: they are required to implement the operation of walk extension in hardware. For walk extension one needs to look-up the page table entries in the memory, and in order to perform an access to the hardware memory (cache) we need registers to provide the inputs or retrieve the outputs of that access. The content of these registers is considered meaningful only if the nested MMU component is currently processing a translation request which involves (!) the corresponding walk register.

$$w_G(mmu) = \begin{cases} \{mmu.w_G\} & \overline{mmu.abort} \wedge \overline{mmu.idle} \\ \emptyset & \text{otherwise} \end{cases}$$

$$w_U(mmu) = \begin{cases} \{mmu.w_U\} & \overline{mmu.abort} \wedge \overline{mmu.idle_U} \\ \emptyset & \text{otherwise} \end{cases}$$

Thus, if the nested MMU stays in its $idle_U$ control state but remains busy, it processes an ordinary translation request (of the guest address), which clearly does not involve the user walk register. In case the nested MMU leaves the $idle_U$ state, there is a nested translation request (of the user address) being processed, which involves both walk registers. Invalidation requests do not trigger the busy signal. They are handled in one cycle and allowed only in cycles in which the nested MMU is not busy (see Sect. 4.2.1).

Using the definitions above we easily express the simulation of the walk registers.

$$\begin{aligned} sim_W(mmu,c) \;\equiv\; & w_U(mmu) \cup w_G(mmu) \subseteq c.tlb \;\wedge \\ & w_U(mmu) \circ w_G(mmu) \subseteq c.tlb^\circ \end{aligned}$$

Note that the second part (for composition) by definition follows from the first part. It was included to i) stress that the composition of hardware walks belongs to the composed TLB and ii) justify the upcoming definitions (for sets of hardware walks). A trivial lemma follows.

Lemma 35.

$$mmu.idle \vee mmu.abort \;\rightarrow\; w_G(mmu) \cup w_U(mmu) = \emptyset$$

Next we specify the MMU configurations in which the specification TLB is stepped. We do it in such a way that the resulting ISA computation simulates the hardware computation. In the next sections we keep using our old abbreviations to save space.

$$\begin{aligned} w_G &\equiv mmu.w_G \\ w_U &\equiv mmu.w_U \end{aligned}$$

5.2.3 Simulation Theorem

Finally we have enough machinery to justify the implementations of hardware units from Chap. 4. Below and elsewhere throughout this section by h (c) and h' (c') we denote the current and the next state configurations of hardware (ISA) resp. Informally

$$h \to h'$$
$$c \to c'$$

where the next state configuration of hardware and ISA are resp. defined by the hardware MMU specification from Sect. 4.2.1 and the general TLB specification from Sect. 3.4. Independently of a particular machine design, the nested MMU changes only in those configurations in which it is queried, i.e., if there is a translation/invalidation query among the operations performed by hardware (see Sect. 5.1). Otherwise, the nested MMU stays unchanged. If requested, the nested MMU generates steps to be performed by the ISA computation, as we described in Sect. 5.2.1. For convenience, on every reference to the MMU hardware configuration below we abbreviate as usual:

$$h.Z \equiv mmu.Z$$
$$Z(h) \equiv Z(mmu).$$

The total number of steps performed within the hardware transition ($h \to h'$) is given by $ns(h)$. From lemma 34 we know

$$ns(h) \leq 1,$$

therefore it suffices to pass $s(h)$ to the ISA computation as input to perform those steps (one step at most). This input is extracted from the hardware computation in configuration h, as we specified above, in Sect. 5.2.1. Having that we can define the next state configuration of ISA:

$$c'.tlb = \begin{cases} \delta_{tlb}(c.tlb, s(h)) & ns(h) > 0 \\ c.tlb & \text{otherwise.} \end{cases}$$

We leave the next state configuration of hardware (h') without a formal definition. One can compose this definition from the formal hardware specification of the nested MMU and the implementation details from Chap. 4. The resulting transition function (δ_{mmu}) would naturally depend on the values coming at the inputs of the component. In what follows we prove correctness of the nested MMU hardware for arbitrary valid sequence of input values.

Lemma 36. *For every valid query sequence qr:*

$$\exists c^0\ \forall t : sim_{tlb}(mmu^t_{qr}, c^t)$$

In order to prove the latter lemma we proceed as follows:

i) since in hardware we store walks both in the TLB cache and walk registers, first in Sect. 5.3 we introduce an auxiliary simulation relation (sim_A) to smoothen these nuances out and make the forthcoming proofs simpler, then
ii) we state an invariant about the user walks stored in hardware ($inv_{\circ}$), which turns out to be crucial in order to show correct simulation of these walks, next
iii) we make the last preparations before we do proofs, namely we show how the walks dropped/added by the invalidation/translation queries in hardware relate to the corresponding walks in ISA, and finally
iv) we prove in Sect. 5.4 that the entire simulation relation for the TLB (sim_{tlb}) is preserved throughout and after execution of invalidation/translation queries and, obviously, in the absence of any queries.

5.3 Developing Formalism

First, we require a counterpart of the definition from [LOP] where we introduced the set of all walks stored in hardware ($walks(h)$). In case of the nested translation we — as usual — distinguish between the hardware guest ($walks_G(h)$) and user ($walks_U(h)$) walks. The following definitions are self-explaining.

$$\begin{aligned} walks_G(h) &= tlb_G(h) \cup w_G(h) \\ walks_U(h) &= tlb_U(h) \cup w_U(h) \circ w_G(h) \end{aligned}$$

Since in the proofs later we argue mostly about these sets of walks, we introduce an auxiliary simulation relation which operates on these sets.

$$\begin{aligned} sim_A(h,c) \equiv\ & walks_G(h) \subseteq c.tlb \ \wedge \\ & walks_U(h) \subseteq c.tlb^{\circ} \end{aligned}$$

Most of the effort will be spent to show that the simulation above holds in the subsequent configurations of hardware (h') and ISA (c'). Once it is done, we extend this simulation to the "entire" one (sim_{tlb}) with a very mild effort. Indeed, a trivial lemma shows that only simulation of the user walk register is missing in that case.

Lemma 37.

$$sim_{tlb}(h,c) \equiv sim_A(h,c) \wedge w_U(h) \subseteq c.tlb$$

Proof of lemma 37. Gradually unfolding definitions we argue as follows.

$$\begin{aligned} sim_{tlb}(h,c) &\equiv sim_T(h,c) \wedge sim_W(h,c) \\ &\equiv tlb_G(h) \subseteq c.tlb \ \wedge \ w_G(h) \cup w_U(h) \subseteq c.tlb \ \wedge \\ &\quad tlb_U(h) \subseteq c.tlb^{\circ} \wedge w_G(h) \circ w_U(h) \subseteq c.tlb^{\circ} \\ &\equiv walks_G(h) \subseteq c.tlb \ \wedge \ w_U(h) \subseteq c.tlb \ \wedge \\ &\quad walks_U(h) \subseteq c.tlb^{\circ} \\ &\equiv sim_A(h,c) \ \wedge \ w_U(h) \subseteq c.tlb \end{aligned}$$

□

5.3.1 Coverage of Hardware Walks

It turns out that the simulation of user walks ($walks_U$) hinges on a slightly more general property. Namely, we claim that for every user walk stored in hardware there is a pair of walks in the specification TLB s.t. in composition walks from the pair form the (hardware) user walk and the page address of the guest walk from the pair is the same as the guest page address of the (hardware) user walk. Formally, we maintain the following invariant.

Invariant 4. *For configurations h and c we claim that the following holds:*

$$\forall w \in walks_U(h)\ \exists w_u, w_g \in c.tlb: \ (w = w_u \circ w_g) \wedge (gpa(w,h) = w_g.pa)$$

In order to have the guest page address defined for all user walks ($walks_U$), we extend the definition s.t. it can be applied to the user walk stored in the user walk register.

Definition 7. *For walk* $w \in w_U(h) \circ w_G(h)$*:*

$$gpa(w,h) = w_G.pa$$

Note, the latter is well defined since

$$w_U(h) \circ w_G(h) \subseteq tlbset(h)$$

occurs only after a write into the TLB. In case the walk was written into the entry i, by construction we have

$$h.tlb.g(i) = w_G.pa.$$

We call the invariant above a *coverage* of the (hardware) user walks, meaning that the walks in hardware are covered by the walks in software, and refer to it in proofs simply by $inv_\circ(h,c)$. As we show below, this invariant trivially implies the desired simulation of the user walks.

To simplify the forthcoming proofs lines, we introduce somewhat more technical notation. We use the following to *select* from set $walks_U$ a user walk which is covered by pair (w_u, w_g) of walks in the sense of invariant 4.

$$walks_U(h)[w_u][w_g] = \{w \in walks_U(h) \mid (w = w_u \circ w_g) \wedge (gpa(w,h) = w_g.pa)\}$$

In the obvious way we extend the notation above for passing sets of walks instead of individual walks. For instance, to select using w_g as a guest walk and all user walks possible, we write:

$$walks_U(h)[*][w_g] \ \equiv \bigcup_{w_u \in K_{uwalk}} walks_U(h)[w_u][w_g].$$

In case we need to consider only those user walks which are available in the specification TLB of configuration c, we similarly write:

$$walks_U(h)[c][w_g] \ \equiv \bigcup_{w_u \in c.tlb} walks_U(h)[w_u][w_g].$$

Naturally, we apply the notation above to select using w_u as a user walk and sets of guest walks, either all possible or only those available in the specification TLB of configuration c. The new notation turns to be very intuitive and easy to use. For instance, invariant 4 can be equivalently rewritten simply as follows.

$$walks_U(h) = walks_U(h)[c][c]$$

Proofs also become shorter, and thus easier to understand. Some of them nearly boil down to simple bookkeeping. We demonstrate this in the proof below, where we show that the coverage ($inv_\circ(h,c)$) implies the simulation of the user walks ($walks_U$).

Lemma 38. *For configurations h and c such that $inv_\circ(h,c)$ holds:*

$$walks_U(h) \subseteq c.tlb^\circ$$

Proof of lemma 38.

$$\begin{aligned}
&walks_U(h)\\
&= walks_U(h)[c][c] \qquad (inv_\circ(h,c))\\
&= \bigcup_{w_u \in c.tlb} \bigcup_{w_g \in c.tlb} walks_U(h)[w_u][w_g] \qquad \text{(notation)}\\
&= \bigcup_{w_u \in c.tlb} \bigcup_{w_g \in c.tlb} \{w \in walks_U(h) \mid (w = w_u \circ w_g) \wedge (gpa(w,h) = w_g.pa)\} \qquad \text{(definition)}\\
&\subseteq \bigcup_{w_u \in c.tlb} \bigcup_{w_g \in c.tlb} \{w \mid w = w_u \circ w_g\}\\
&= c.tlb^\circ \qquad \text{(definition)}
\end{aligned}$$

□

The notation above can be easily generalized for arbitrary sets of user walks. That helps us to reduce the length of the arguments in the proofs below. For instance, first we show the following.

Lemma 39. *For any pair of sets $\mathcal{W}_1$ and $\mathcal{W}_2$ (of walks) such that*

$$\mathcal{W}_1 \subseteq \mathcal{W}_2[\mathtt{w_u}][\mathtt{w_g}]$$

we claim

$$\mathcal{W}_1 = \mathcal{W}_1[\mathtt{w_u}][\mathtt{w_g}]$$

where $\mathtt{w_u}$ and $\mathtt{w_g}$ are two arbitrary sets of resp. user and guest software walks.

Proof of lemma 39. From the assumptions we naturally obtain

$$\begin{aligned} \mathcal{W}_1 &\subseteq \mathcal{W}_2[w_u][w_g] \\ &\subseteq \mathcal{W}_2[*][*] \\ &= \mathcal{W}_2 \end{aligned} \tag{5}$$

which allows us to derive

$$\begin{aligned} \mathcal{W}_1 &\subseteq \mathcal{W}_2[w_u][w_g] \cap \mathcal{W}_1 \\ &= (\mathcal{W}_2 \cap \mathcal{W}_1)[w_u][w_g] \qquad \text{(equation 5)} \\ &= \mathcal{W}_1[w_u][w_g]. \end{aligned}$$

The claim follows, since for any set of walks $\mathcal{W}$ by definition we clearly have

$$\mathcal{W}[w_u][w_g] \subseteq \mathcal{W}. \qquad \square$$

Now using the latter lemma we argue that the user walks from the *hardware TLB* are naturally covered by the walks in the specification in case invariant 4 is maintained.

Lemma 40. *For configurations h and c such that $inv_\circ(h,c)$ holds:*

$$tlb_U(h) = tlb_U(h)[c][c]$$

Proof of lemma 40. Using invariant 4 we derive

$$\begin{aligned} tlb_U(h) &\subseteq walks_U(h) \qquad \text{(definition)} \\ &= walks_U(h)[c][c] \qquad (inv_\circ(h,c)) \end{aligned}$$

and the claim follows by lemma 39. □

5.3.2 Dropping Translations

Recall, in Sect. 3.4.2 we introduced the sets of invalidated ($\mathcal{I_V}$) and incomplete ($\mathcal{I_C}$) walks, which comprise translations that are dropped on the invalidating TLB steps. In particular, set $\mathcal{I_V}$ was defined s.t. it accumulates either the guest or the user walks, depending resp. on whether the guest or the user address is invalidated. Counterparts of these sets in hardware were defined in Sect. 4.1.1, in which we formalized the semantics of the hardware TLB. For the upcoming argument about correctness of the invalidating TLB steps we need to relate the sets defined for the specification with their hardware counterparts.

In the following lemma we show that in case a guest address is invalidated, the hardware drops all guest walks that fall into the sets of invalidated and incomplete walks.

Lemma 41. *On invalidation of an individual guest address*

$$invlpg(h) \wedge inva(h) \in A_G$$

we claim the following holds:

$$invset(h) = tlb_G(h) \cap \mathcal{I_V}(s(h)) \tag{1}$$

$$incset(h) = tlb_G(h) \cap \mathcal{I_C}(s(h)) \tag{2}$$

Proof of lemma 41.1.

$$\begin{aligned} &tlb_G(h) \cap \mathcal{I_V}(s(h)) \\ &= tlb_G(h) \cap \{w \mid w.upa = s(h).upa\} \qquad \text{(definition)} \\ &= tlb_G(h) \cap \{w \mid w.upa = inva(h)\} \qquad \text{(stepping)} \\ &= tlbset(h) \cap \{w \mid w.upa = inva(h)\} \qquad \text{(assumptions)} \\ &= invset(h) \qquad \text{(definition)} \end{aligned}$$ □

Proof of lemma 41.2.

$$\begin{aligned}
&tlb_G(h) \cap \mathcal{I}_{\mathcal{C}}(s(h))\\
&= tlb_G(h) \cap \{w \mid w.as = s(h).upa.as \wedge \overline{w.\ell[0]}\} \qquad \text{(definition)}\\
&= tlb_G(h) \cap \{w \mid w.as = inva(h).as \wedge \overline{w.\ell[0]}\} \qquad \text{(stepping)}\\
&= tlbset(h) \cap \{w \mid w.as = inva(h).as \wedge \overline{w.\ell[0]}\} \qquad \text{(assumptions)}\\
&= incset(h) \qquad \text{(definition)}
\end{aligned}$$

□

In addition, the hardware drops all user walks that are covered by the sets of invalidated and incomplete (guest) walks. This is done in order to preserve the invariant, in accordance with intuition from Sect. 4.1. Otherwise, in case a user address is invalidated, the hardware drops all (user) walks that are covered by the sets of invalidated and incomplete (user) walks. We formalize the latter two results in the following lemma. In order to save space, for $\mathcal{X} \in \{\mathcal{V}, \mathcal{C}, \mathcal{D}\}$ we sometimes abbreviate:

$$\mathcal{I}_{\mathcal{X}} = \mathcal{I}_{\mathcal{X}}(s(h)).$$

Lemma 42. *On invalidation of an individual address*

$$invlpg(h)$$

we claim the following holds:

$$\begin{aligned}
inva(h) \in A_G &\rightarrow & ragset(h) &\supseteq tlb_U(h)[c][\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}] \qquad (1)\\
inva(h) \in A_U &\rightarrow & invset(h) \cup incset(h) &\supseteq tlb_U(h)[\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}][c] \qquad (2)
\end{aligned}$$

Proof of lemma 42.1. In the proof below we argue using part (1) of invariant 3 as follows. When ragged walk

$$w = w_u \circ w_i$$

was composed from certain incomplete guest walk in hardware, the latter guest walk (w_i) was contained in the hardware TLB.

$$w \in tlb_U(h) \rightarrow \exists \tilde{h} : w_i \in tlb_G(\tilde{h})$$

For all incomplete walks contained in the hardware TLB we know from the invariant they are faulty. Using definition of the walk composition we conclude the following.

$$w_i \in \mathcal{I}_{\mathcal{C}}(s(h)) \rightarrow f(w_i) \wedge w.f_g \qquad (6)$$

Having this we proceed to show the claim.

$$\begin{aligned}
&tlb_U(h)[c][\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}]\\
&\subseteq tlb_U(h)[*][\mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}]\\
&= \bigcup_{w_i \in \mathcal{I}_{\mathcal{V}} \cup \mathcal{I}_{\mathcal{C}}} \{w \in tlb_U(h) \mid \exists w_u : (w = w_u \circ w_i) \wedge (gpa(w,h) = w_i.pa)\} \qquad \text{(notation)}\\
&\subseteq tlb_U(h) \cap \bigcup_{w_i \in \mathcal{I}_{\mathcal{V}}} \{w \mid (w.vm = w_i.vm) \wedge (gpa(w,h) = w_i.pa)\} \cup\\
&\quad tlb_U(h) \cap \bigcup_{w_i \in \mathcal{I}_{\mathcal{C}}} \{w \mid (w.vm = w_i.vm) \wedge w.f_g\} \qquad \text{(definition, equation 6)}\\
&= tlb_U(h) \cap \{w \mid (w.vm = s(h).upa.vm) \wedge ((gpa(w,h) = s(h).upa.pa) \vee w.f_g)\} \qquad \text{(definition)}\\
&= tlb_U(h) \cap \{w \mid (w.vm = inva(h).vm) \wedge ((gpa(w,h) = inva(h).pa) \vee w.f_g)\} \qquad \text{(stepping)}\\
&= tlbset(h) \cap \{w \mid (w.upa \in A_U(inva(h).vm)) \wedge ((gpa(w,h) = inva(h).pa) \vee w.f_g)\}\\
&= ragset(h) \qquad \text{(definition)}
\end{aligned}$$

□

Proof of lemma 42.2.

$$\begin{aligned}
& tlb_U(h)[\mathcal{I_V} \cup \mathcal{I_C}][c] \\
& \subseteq tlb_U(h)[\mathcal{I_V} \cup \mathcal{I_C}][*] \\
& = \bigcup_{w_i \in \mathcal{I_V} \cup \mathcal{I_C}} \{w \in tlb_U(h) \mid \exists w_g : (w = w_i \circ w_g) \ \wedge \ (gpa(w,h) = w_g.pa)\} \qquad \text{(notation)} \\
& \subseteq tlb_U(h) \ \cap \bigcup_{w_i \in \mathcal{I_V}} \{w \mid w.upa = w_i.upa\} \ \cup \\
& \quad tlb_U(h) \ \cap \bigcup_{w_i \in \mathcal{I_C}} \{w \mid (w.as = w_i.as) \wedge (w.\ell = w_i.\ell)\} \qquad \text{(definition)} \\
& \subseteq tlb_U(h) \cap \{w \mid (w.upa = s(h).upa) \vee (w.as = s(h).upa.as) \wedge \overline{w.\ell[0]}\} \qquad \text{(definition)} \\
& = tlb_U(h) \cap \{w \mid (w.upa = inva(h)) \vee (w.as = inva(h).as) \wedge \overline{w.\ell[0]}\} \qquad \text{(stepping)} \\
& = tlbset(h) \cap \{w \mid (w.upa = inva(h)) \vee (w.as = inva(h).as) \wedge \overline{w.\ell[0]}\} \\
& = invset(h) \ \cup \ incset(h) \qquad \text{(definition)} \qquad \square
\end{aligned}$$

In the following lemma we formalize the last result about translations dropped from the hardware TLB on invalidating steps. So, in case a virtual machine is invalidated, all (user) walks that are covered by the set of dropped (user) walks are dropped from the hardware TLB.

Lemma 43. *On invalidation of a virtual machine*

$$vmflush(h)$$

we claim the following holds:

$$invset(h) \supseteq tlb_U(h)[\mathcal{I_D}][c]$$

Proof of lemma 43.

$$\begin{aligned}
& tlb_U(h)[\mathcal{I_D}][c] \\
& \subseteq tlb_U(h)[\mathcal{I_D}][*] \\
& = \bigcup_{w_i \in \mathcal{I_D}} \{w \in tlb_U(h) \mid \exists w_g : (w = w_i \circ w_g) \ \wedge \ (gpa(w,h) = w_g.pa)\} \qquad \text{(notation)} \\
& \subseteq tlb_U(h) \ \cap \bigcup_{w_i \in \mathcal{I_D}} \{w \mid w.upa = w_i.upa\} \qquad \text{(definition of } \circ) \\
& \subseteq tlb_U(h) \cap \{w \mid w.upa \in A_U(s(h).vm)\} \qquad \text{(definition)} \\
& = tlb_U(h) \cap \{w \mid w.upa \in A_U(inva(h).vm)\} \qquad \text{(stepping)} \\
& = tlbset(h) \cap \{w \mid w.upa \in A_U(inva(h).vm)\} \\
& = invset(h) \qquad \text{(definition)} \qquad \square
\end{aligned}$$

Also we argue that *after* the invalidation queries the content of both walk registers is not meaningful.

Lemma 44.

$$tdrop(h) \ \rightarrow \ w_G(h') \cup w_U(h') = \emptyset$$

Proof of lemma 44. By case split on whether the MMU is idle in configuration h:

- $h.idle = 1$. In case the idle MMU is not requested for translation, it remains idle.

$$h.idle \wedge /h.treq \ \rightarrow \ h'.idle$$

- $h.idle = 0$. If the ongoing translation was not aborted yet, it is aborted.

$$/h.abort \wedge /h.idle \wedge /h.treq \ \rightarrow \ h'.abort$$

  Otherwise, the abort signal stays high in configuration h'.

$$h.abort \wedge /h.idle \ \rightarrow \ h'.abort$$

In all considered cases the claim follows by lemma 35. □

5.3.3 Adding Translations

In Sect. 3.4.2 we specified the walks ($w(x)$) which are added to the specification TLB on resp. guest ($tadd_G$) and user ($tadd_U$) TLB steps. For the upcoming argument for correctness of the TLB steps we need to show that these walks coincide with the inputs of the corresponding walk registers in hardware.

Lemma 45. *On addition of translations*

$$tadd(h)$$

we claim the following holds:

$$tadd_G(h) \rightarrow w(s(h)) = w_g(h) \tag{1}$$
$$tadd_U(h) \rightarrow w(s(h)) = w_u(h) \tag{2}$$

Recall, by definition of the stepping function we have:

$$tadd_X(h) \rightarrow s(h) = \begin{cases} (winit, upa_X(h), h.pto_X) & winit_X(h) \\ (wext, w_X, pte_X(h)) & \text{otherwise.} \end{cases}$$

Proof of lemma 45.1.

$$\begin{aligned} w(s(h)) &= \begin{cases} winit(upa, pto) & s(h) = (winit, upa, pto) \\ wext(w, pte) & s(h) = (wext, w, pte) \end{cases} && \text{(definition)} \\ &= \begin{cases} winit(upa_G(h), h.pto_G) & winit_G(h) \\ wext(w_G, pte_G(h)) & \text{otherwise} \end{cases} && \text{(stepping)} \\ &= w_g(h) \qquad \text{(construction)} \end{aligned}$$

□

Proof of lemma 45.2.

$$\begin{aligned} w(s(h)) &= \begin{cases} winit(upa, pto) & s(h) = (winit, upa, pto) \\ wext(w, pte) & s(h) = (wext, w, pte) \end{cases} && \text{(definition)} \\ &= \begin{cases} initw(upa_U(h), h.pto_U) & winit_U(h) \\ wext(w_U, pte_U(h)) & \text{otherwise} \end{cases} && \text{(stepping)} \\ &= w_u(h) \qquad \text{(construction)} \end{aligned}$$

□

5.4 Correctness Proof

Finally, we proceed to the proof of lemma 36. The proof is obviously by induction on the number of hardware cycles t. For the base case ($t = 0$) we consider

$$c^0.tlb = \emptyset$$

and argue as follows. Simulation relation for the TLB content (sim_T) holds trivially.

$$tlb_G(h^0) = \emptyset \subseteq c^0.tlb$$
$$tlb_U(h^0) = \emptyset \subseteq c^0.tlb^\circ$$

The simulation relation for the walk registers (sim_W) holds by lemma 35. Invariant 4 as well holds trivially after the initialization of hardware.

$$walks_U(h^0) \; = \; \emptyset \; = \; walks_U(h^0)[c^0][c^0]$$

For the induction step ($t \rightarrow t+1$) we split cases on the type of query processed by hardware in cycle t:

- $qr[t].treq$ — translation queries — new translation *can be* added ($tadd_X(t)$),
- $qr[t].ireq$ — invalidation queries — old translation(s) are dropped ($tdrop(t)$), and
- otherwise — void queries — no steps are performed in cycle t.

We cover each type in a dedicated section below. For convenience we abbreviate configurations of hardware (h) and ISA (c) in cycles t and $t+1$. For $z \in \{h,c\}$ we abbreviate as usual:

$$(z,z') \;\equiv\; (z^t,z^{t+1}).$$

5.4.1 Void Queries

We consider the most simple case first: we show that in the absence of any requests, the configuration of the MMU component stays unchanged, which implies that the simulation of the MMU content as well as the invariant 4 — about coverage of the user walks — are maintained. Below we make our arguments formal.

Simulation

From the hardware construction — automata of the nested MMU, Sect. 4.2.2 — we first argue that the content of the walk registers is not meaningful in configuration h'.

$$w_U(h') \cup w_G(h') = \emptyset$$

In order to show the claim we cover the following cases:

i) loss of the translation request (aborting translations),
ii) clearing the pending abort signal (restoring after aborts), and
iii) waiting for a query/restore.

In all cases above we derive directly from the control mechanisms:

- aborting translations ($h.abort.set$)
$$h.abort.set \;\rightarrow\; h'.abort$$
- restoring after aborts ($h.abort.clr$)
$$h.abort.clr \;\rightarrow\; h'.idle$$
- waiting for a query/restore ($/h.abort.set \wedge /h.abort.clr$)
$$h.abort \oplus h.idle \;\rightarrow\; h'.abort \oplus h'.idle$$

According to the definition it remains to show

$$sim_T(h',c').$$

From the absence of queries we know:

i) no translation is performed in configuration h ($/tadd(h) \wedge /h.tlb.store$), and
ii) no invalidation is performed in configuration h ($/tdrop(h) \wedge /h.tlb.inval$).

Therefore, the hardware TLB does not change

$$/h.tlb.store \wedge /h.tlb.inval \;\rightarrow\; h'.tlb = h.tlb$$

and the specification TLB does not change either.

$$/tadd(h) \wedge /tdrop(h) \;\rightarrow\; c' = c$$

Thus, simulation of the TLB content (sim_T) we conclude from the arguments above.

$$\begin{aligned} tlb_G(h') = tlb_G(h) &\subseteq c.tlb \;\; = c'.tlb \\ tlb_U(h') = tlb_U(h) &\subseteq c.tlb^{\circ} = c'.tlb^{\circ} \end{aligned}$$

Invariant

Next we argue about the invariant. As we have already shown above, the content of the walk registers is not meaningful in configuration h'. Therefore we have

$$w_U(h') \circ w_G(h') = \emptyset$$

and proceed to show the following.

$$\begin{aligned} walks_U(h') &= tlb_U(h') \cup w_U(h') \circ w_G(h') && \text{(definition)} \\ &= tlb_U(h) && \text{(construction)} \\ &= tlb_U(h)[c][c] && \text{(lemma 40)} \\ &= tlb_U(h)[c'][c'] && (c' = c) \end{aligned}$$

The claim follows by lemma 39. Note that in the absence of any requests, as well as in the cycles in which the nested MMU is busy waiting (for the memory system), function gpa does not change the values for the walks from set $walks_U$.

5.4.2 Translation Queries

Here we consider the translation queries and show correctness of the TLB steps — of walk initialization and extension. We need to make sure that in all hardware configurations which can be encountered during execution of the translation queries the content of the nested MMU is correctly simulated and all user walks are covered by the walks in the specification TLB.

Guest Simulation

First we establish simulation of the guest walks ($walks_G$) as it was formulated in relation sim_A (see Sect. 5.2.3). We split the proof of

$$walks_G(h') \subseteq c'.tlb$$

into the following four cases:

i) clocking of the walk registers (adding walks),
ii) writing walks from the walk registers into the TLB cache,
iii) writing walks from the TLB back into the walk registers, and
iv) waiting for the memory (cycles in which the previous two cases do not apply).

- adding walks ($tadd(h)$). We split further into two cases, depending on the type (guest or user) of the walk added.
  - adding a guest walk ($tadd_G(h)$).

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &= tlb_G(h) \cup \{w_g(h)\} && \text{(construction)} \\ &\subseteq walks_G(h) \cup \{w_g(h)\} && \text{(definition)} \\ &\subseteq c.tlb \cup \{w_g(h)\} && (sim_A(h,c)) \\ &= c.tlb \cup \{w(s(h))\} && \text{(lemma 45.1)} \\ &= c'.tlb && \text{(specification)} \end{aligned}$$

  - adding a user walk ($tadd_U(h)$).

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &= tlb_G(h) \cup w_G(h) && \text{(construction)} \\ &= walks_G(h) && \text{(definition)} \\ &\subseteq c.tlb && (sim_A(h,c)) \\ &\subseteq c.tlb \cup \{w(s(h))\} \\ &= c'.tlb && \text{(specification)} \end{aligned}$$

- store a walk into the hardware TLB ($/tadd(h)$). We again split cases further, but this time on whether a guest walk or a composition of walks is written.
  - writing a guest walk ($write\text{-}tlb_G(h)$).

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &\subseteq (tlb_G(h) \cup w_G(h)) \cup w_G(h) && \text{(construction)} \\ &= walks_G(h) && \text{(definition)} \\ &\subseteq c.tlb && (sim_A(h,c)) \\ &= c'.tlb && (c' = c) \end{aligned}$$

  - writing a composition of walks ($write\text{-}tlb_U(h)$).

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &\subseteq tlb_G(h) \cup w_G(h) && \text{(construction)} \\ &= walks_G(h) && \text{(definition)} \\ &\subseteq c.tlb && (sim_A(h,c)) \\ &= c'.tlb && (c' = c) \end{aligned}$$

  Note that in order to fit the walk which is written, the hardware evicts some valid entry (stored walk) in case the TLB cache is full. Also, due to the invariant (about uniqueness of walks with respect to translated addresses) maintained by our hardware construction (see Sect. 5.2.3), a valid walk stored in hardware is overwritten each time one attempts to store another walk for the same universal page address.
- writing the walk register with a walk from the hardware TLB ($/tadd(h)$). The latter concerns only the guest walk register, which is written in state *nested-call* with a guest walk from the TLB each time the requested translation was present in the cache (TLB) before the simple translation started.

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &= tlb_G(h) \cup \{h.wout\} && \text{(construction)} \\ &\subseteq walks_G(h) && \text{(definition)} \\ &\subseteq c.tlb && (sim_A(h,c)) \\ &= c'.tlb && (c' = c) \end{aligned}$$

- busy waiting ($/tadd(h)$). Finally, we argue that in the hardware cycles in which the TLB and the walk registers are not written (in the absence of the TLB steps ($tadd(h)$) and outside of the control state $write\text{-}tlb_X(h)$), the simulation of the guest walks is preserved.

$$\begin{aligned} walks_G(h') &= tlb_G(h') \cup w_G(h') && \text{(definition)} \\ &= tlb_G(h) \cup w_G(h) && \text{(construction)} \\ &= walks_G(h) && \text{(definition)} \\ &\subseteq c.tlb && (sim_A(h,c)) \\ &= c'.tlb && (c' = c) \end{aligned}$$

Invariant

Next we show that invariant 4 is maintained over the translation queries. In the proof we implicitly use a hardware property that values of function *gpa* are persistent for the user walks from the TLB (tlb_U). Both if the hardware cache ($h.tlb$) is written or not, the values "associated" with the hardware walks do not change.

We split the proof of

$$inv_\circ(h', c')$$

into the same cases as the proof above: i) adding walks, ii)-iii) writing walks, and iv) busy waiting.

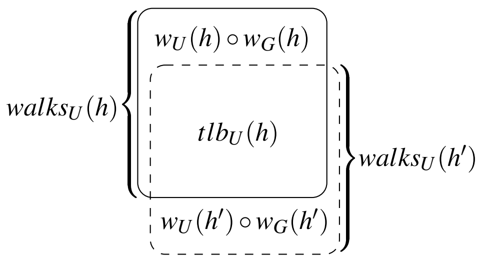


Fig. 26: Changes in the set of user walks ($walks_U$) on adding of walks

- adding walks. For convenience we introduce an abbreviation

$$\{w'\} = w_U(h') \circ w_G(h')$$

to refer to the newly added walk. As depicted in Fig. 26, the set of user walks ($walks_U$) in configuration h' consists of i) the user walks in the TLB cache, which remain unchanged, and ii) the newly added walk (w'):

$$walks_U(h') = tlb_U(h') \cup \{w'\}.$$

First, we show that the coverage is maintained for the user walks from the TLB.

$$\begin{aligned} tlb_U(h') &= tlb_U(h) && \text{(construction)} \\ &\subseteq walks_U(h) \cap walks_U(h') && \text{(definition)} \\ &= walks_U(h)[c][c] \cap walks_U(h') && (inv_\circ(h,c)) \\ &\subseteq walks_U(h')[c][c] \\ &\subseteq walks_U(h')[c'][c'] && \text{(specification)} \end{aligned}$$

From the definition of the guest page address for the newly added walk we have:

$$gpa(w',h') = (\varepsilon\ w_G(h')).pa.$$

Therefore, walk w' is covered by walks $w_U(h')$ and $w_G(h')$, which are contained in the specification TLB of configuration c'. We justify the latter below.

  - adding a guest walk.

$$\begin{aligned} \{w'\} &\subseteq walks_U(h')[w_U(h')][w_G(h')] && \text{(definition)} \\ &= walks_U(h')[w_U(h)][w_g(h)] && \text{(construction)} \\ &\subseteq walks_U(h')[c][w(s(h))] && (sim_W(h,c);\text{ lemma 45.1}) \\ &\subseteq walks_U(h')[c'][c'] && \text{(specification)} \end{aligned}$$

  - adding a user walk.

$$\begin{aligned} \{w'\} &\subseteq walks_U(h')[w_U(h')][w_G(h')] && \text{(definition)} \\ &= walks_U(h')[w_u(h)][w_G(h)] && \text{(construction)} \\ &\subseteq walks_U(h')[w(s(h))][c] && (\text{lemma 45.2; } sim_W(h,c)) \\ &\subseteq walks_U(h')[c'][c'] && \text{(specification)} \end{aligned}$$

- store a walk into the hardware TLB.

$$\begin{aligned} walks_U(h') &= tlb_U(h') \cup w_U(h') \circ w_G(h') && \text{(definition)} \\ &\subseteq (tlb_U(h) \cup w_U(h) \circ w_G(h)) \cup w_U(h) \circ w_G(h) && \text{(construction)} \\ &= walks_U(h) && \text{(definition)} \\ &= walks_U(h)[c][c] && (inv_\circ(h,c)) \\ &= walks_U(h)[c'][c'] && (c' = c) \end{aligned}$$

Note, the set of user walks ($walks_U$) can only decrease after a write of the TLB.

- writing the guest walk register with a walk from the hardware TLB ($/tadd(h)$). Using notation from the first case above, we first argue that coverage of the user walks from the TLB is maintained; the proof stays literally the same. However we need to update the argument for the newly obtained composition of walks.

$$\begin{aligned}\{w'\} &\subseteq walks_U(h')[w_U(h')][w_G(h')] && \text{(definition)}\\ &= walks_U(h')[w_U(h)][h.wout] && \text{(construction)}\\ &\subseteq walks_U(h')[c][c] && (sim_W(h,c))\\ &= walks_U(h')[c'][c'] && (c'=c)\end{aligned}$$

- busy waiting. Nothing changes in hardware, therefore nothing to show as before.

$$\begin{aligned}walks_U(h') &= walks_U(h) && \text{(construction)}\\ &= walks_U(h)[c][c] && (inv_\circ(h,c))\\ &= walks_U(h)[c'][c'] && (c'=c)\end{aligned}$$

Note, in all cases the claim follows by lemma 39.

User Simulation

Finally, we show that simulation sim_{tlb} is preserved over the translation queries.

$$sim_{tlb}(h',c')$$

Previously we have shown

$$walks_G(h') \subseteq c'.tlb$$

and

$$inv_\circ(h',c').$$

The latter by lemma 38 gives

$$walks_U(h') \subseteq c'.tlb^\circ.$$

Therefore, we have $sim_A(h',c')$ and by lemma 37 it remains to show

$$w_U(h') \subseteq c'.tlb.$$

We consider only those cycles in which the walk registers are clocked (updated). In other cycles their values do not change, and the proof is trivial. We split cases depending on the register which is clocked. By lemma 32 this register is unique.

- updating the guest walk register ($tadd_G(h)$).

$$\begin{aligned}w_U(h') &= w_U(h) && \text{(construction)}\\ &\subseteq c.tlb && \text{(lemma 37)}\\ &\subseteq c.tlb \cup \{w(s(h))\}\\ &= c'.tlb && \text{(specification)}\end{aligned}$$

- updating the user walk register ($tadd_U(h)$).

$$\begin{aligned}w_U(h') &= \{w_u(h)\} && \text{(construction)}\\ &= \{w(s(h))\} && \text{(lemma 45.2)}\\ &\subseteq c.tlb \cup \{w(s(h))\}\\ &= c'.tlb && \text{(specification)}\end{aligned}$$

5.4.3 Invalidation Queries

Lastly, we consider the invalidation queries and, respectively, show correctness of steps invalidating the TLB content. In contrast to the translation queries, which in general take multiple hardware cycles, our MMU implementation allows to invalidate all specified translations in a single cycle. This means, we do not have to split cases depending on particular action performed by the nested MMU while it serves an invalidation query. Instead, we split the proofs on specifics of the query served. Our proof goals remain unchanged: we are to show that simulation sim_{tlb} is preserved and invariant 4 is maintained.

Guest Simulation

As before, we start with the simulation of the guest walks. From lemma 44 we have

$$walks_G(h') = tlb_G(h')$$

therefore it suffices to show

$$tlb_G(h') \subseteq c'.tlb.$$

We split cases as follows:

- dropping individual walks ($invlpg(h)$). As before, we split cases further depending on the type (guest or user) of the walks dropped.
  - dropping guest walks ($inva(h) \in A_G$).

$$\begin{aligned} tlb_G(h') &= tlb_G(h) \setminus (invset(h) \cup incset(h)) \qquad \text{(construction)} \\ &= tlb_G(h) \setminus (tlb_G(h) \cap (\mathcal{I_V} \cup \mathcal{I_C})) \qquad \text{(lemma 41)} \\ &= tlb_G(h) \setminus (\mathcal{I_V} \cup \mathcal{I_C}) \\ &\subseteq c.tlb \setminus (\mathcal{I_V} \cup \mathcal{I_C}) \qquad (sim_A(h,c)) \\ &= c'.tlb \qquad \text{(specification)} \end{aligned}$$

  - dropping user walks ($inva(h) \in A_U$).

$$\begin{aligned} tlb_G(h') &= tlb_G(h) \setminus (invset(h) \cup incset(h)) \qquad \text{(construction)} \\ &\subseteq tlb_G(h) \setminus tlb_U(h)[\mathcal{I_V} \cup \mathcal{I_C}][c] \qquad \text{(lemma 42.2)} \\ &= tlb_G(h) \\ &= tlb_G(h) \setminus (\mathcal{I_V} \cup \mathcal{I_C}) \\ &\subseteq c.tlb \setminus (\mathcal{I_V} \cup \mathcal{I_C}) \qquad (sim_A(h,c)) \\ &= c'.tlb \qquad \text{(specification)} \end{aligned}$$

- dropping virtual machines ($vmflush(h)$).

$$\begin{aligned} tlb_G(h') &= tlb_G(h) \setminus invset(h) \qquad \text{(construction)} \\ &\subseteq tlb_G(h) \setminus walks_U(h)[\mathcal{I_D}][c] \qquad \text{(lemma 43)} \\ &= tlb_G(h) \\ &= tlb_G(h) \setminus \mathcal{I_D} \\ &\subseteq c.tlb \setminus \mathcal{I_D} \qquad (sim_A(h,c)) \\ &= c'.tlb \qquad \text{(specification)} \end{aligned}$$

- dropping all walks ($flush(h)$). The simulation of the guest walks holds trivially.

$$\begin{aligned} tlb_G(h') &= \emptyset \qquad \text{(construction)} \\ &= c'.tlb \qquad \text{(specification)} \end{aligned}$$

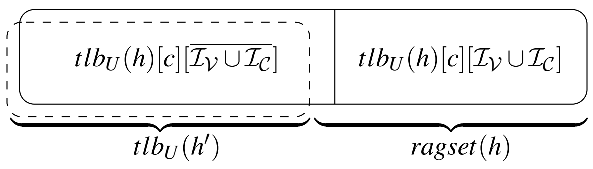

$tlb_U(h)[c][\overline{\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}}]$
$tlb_U(h)[c][\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}]$
$tlb_U(h')$
$ragset(h)$

Fig. 27: Changes in the set of user walks ($walks_U$) on dropping of guest walks

Invariant

Next we show that invariant 4 is maintained over the invalidation queries. Again, since from lemma 44 we have

$$walks_U(h') = tlb_U(h')$$

together with lemma 39 it suffices to show

$$tlb_U(h') = tlb_U(h)[c'][c'].$$

We split the proof into the same cases as the proof above: i) dropping individual walks, ii) dropping virtual machines, and iii) dropping all walks.

- dropping individual walks ($invlpg(h)$). Note, in case one invalidates guest walks (the first case below), the hardware also drops the set of ragged walks according to the hardware specification from Sect. 4.2.1.
  - dropping guest walks ($inva(h) \in A_G$). For more intuition we refer to Fig. 27.

$$\begin{aligned}
tlb_U(h') &= tlb_U(h) \setminus ragset(h) \qquad \text{(construction)}\\
&\subseteq tlb_U(h)[c][c] \setminus tlb_U(h)[c][\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}] \qquad \text{(lemma 40; lemma 42.1)}\\
&= (tlb_U(h)[c][\overline{\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}}] \cup tlb_U(h)[c][\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}]) \setminus tlb_U(h)[c][\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}]\\
&\subseteq tlb_U(h)[c][\overline{\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}}]\\
&= tlb_U(h)[c'][c'] \qquad \text{(specification)}
\end{aligned}$$

  - dropping user walks ($inva(h) \in A_U$).

$$\begin{aligned}
tlb_U(h') &= tlb_U(h) \setminus (invset(h) \cup incset(h)) \qquad \text{(construction)}\\
&\subseteq tlb_U(h)[c][c] \setminus tlb_U(h)[\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}][c] \qquad \text{(lemma 40; lemma 42.2)}\\
&= (tlb_U(h)[\overline{\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}}][c] \cup tlb_U(h)[\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}][c]) \setminus tlb_U(h)[\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}][c]\\
&\subseteq tlb_U(h)[\overline{\mathcal{I}_\mathcal{V} \cup \mathcal{I}_\mathcal{C}}][c]\\
&= tlb_U(h)[c'][c'] \qquad \text{(specification)}
\end{aligned}$$

- dropping virtual machines ($vmflush(h)$).

$$\begin{aligned}
tlb_U(h') &= tlb_U(h) \setminus invset(h) \qquad \text{(construction)}\\
&\subseteq tlb_U(h)[c][c] \setminus tlb_U(h)[\mathcal{I}_\mathcal{D}][c] \qquad \text{(lemma 40; lemma 43)}\\
&= (tlb_U(h)[\overline{\mathcal{I}_\mathcal{D}}][c] \cup tlb_U(h)[\mathcal{I}_\mathcal{D}][c]) \setminus tlb_U(h)[\mathcal{I}_\mathcal{D}][c]\\
&\subseteq tlb_U(h)[\overline{\mathcal{I}_\mathcal{D}}][c]\\
&= tlb_U(h)[c'][c'] \qquad \text{(specification)}
\end{aligned}$$

- dropping all walks ($flush(h)$). The invariant for the user walks holds trivially.

$$\begin{aligned}
tlb_U(h') &= \emptyset \qquad \text{(construction)}\\
&= tlb_U(h)[c'][c'] \qquad \text{(specification)}
\end{aligned}$$

User Simulation

We finish the section showing that simulation sim_{tlb} is preserved over the invalidation queries.

$$sim_{tlb}(h',c')$$

This is a counterpart of the corresponding result about the translation queries. Hence, the proof below follows the same pattern as the respective proof above. Previously we have shown

$$walks_G(h') \subseteq c'.tlb$$

and

$$inv_{\circ}(h',c').$$

The latter by lemma 38 gives

$$walks_U(h') \subseteq c'.tlb^{\circ}.$$

Therefore, we have $sim_A(h',c')$ and by lemma 37 it remains to show

$$w_U(h') \subseteq c'.tlb.$$

which we obtain trivially by lemma 44.

This finishes the proof of lemma 36 and so the chapter on correctness of the nested MMU implementation as well. In the forthcoming chapters we demonstrate how to prove correctness of a simple sequential — yet not completely trivial — machine capable to perform the nested address translation.

Part III

Single-Core MIPS with NAT

6

Sequential Processor with Nested MMUs

The content of this chapter is split into sections as follows. In Sect. 6.1 we specify which components are involved and how these component are interconnected. In order to interconnect the components we introduce several implementation registers together with a small state machine to generate the control signals. Then, in Sect. 6.2 we interconnect the cache memory system and identify the accesses performed by the hardware computation. In Sect. 6.3 we finish the chapter with showing that the presented sequential implementation is live.

6.1 Sequential Processor

To compose the machine we are interested in, obviously, one requires: i) a sequential processor core, ii) two hardware units performing the nested translation for both, the instruction and effective address, and iii) a sequentially consistent shared memory. We resp. take: i) the sequential processor core from [LOP] (with two delayed program counters[1]), ii) two nested MMUs, formally specified and constructed in Chap. 4, and iii) the cache memory system from [KMP14]. The latter memory system we connect to the processor core and the MMUs at the four cache interfaces, as depicted in Fig. 28.

Note, we did not formulate any conditions for the translation inputs of the MMUs (dotted paths in Fig. 28). The reason for that is as follows. If we consider mmu_I, the data for translation are of course coming from the PCs (program counters). For instance, in the pipelined machines — depending on the fullness of the pipe — mmu_I would get the data from one of the three PC registers. PCs of the non-sequential machines are computed speculatively, which means they take wrong values in case of mis-speculation (e.g., on *eret*). Often these (mis-speculated) values do not even belong to the address range of program that is executed. Which in turn means that *translation steps can be performed for addresses that are never used by a source program.* Getting back to our non-deterministic semantics from Chap. 3, one notices that the latter behavior is completely legitimate.

On the other hand, the dotted connections from the processor core to the MMUs are necessary to satisfy a crucial (and very reasonable) guard condition in the specification: *the walk passed to the core has to match the universal address that the core is trying to access* (see Sect. 3.3.4). This condition expresses the essence of the address translation.

After all, the fact that we use these particular data (from the PCs) to start the process of address translation — speculative in certain machines — is nothing more than one out of many possibilities permitted in the specification (even though it might be an intuitive and elegant design

[1] Since in Chap. 8 we consider a seven stage pipelined machine (with two delay slots), here we immediately introduce the corresponding components (delayed PCs) in order to save time later. Thus, for the simpler sequential machine we can introduce, e.g., special purpose register file with appropriate number of exception PCs and use this hardware in the pipelined machine without changes. Moreover, this allows us to reuse certain arguments in the later chapters and focus on more relevant things.

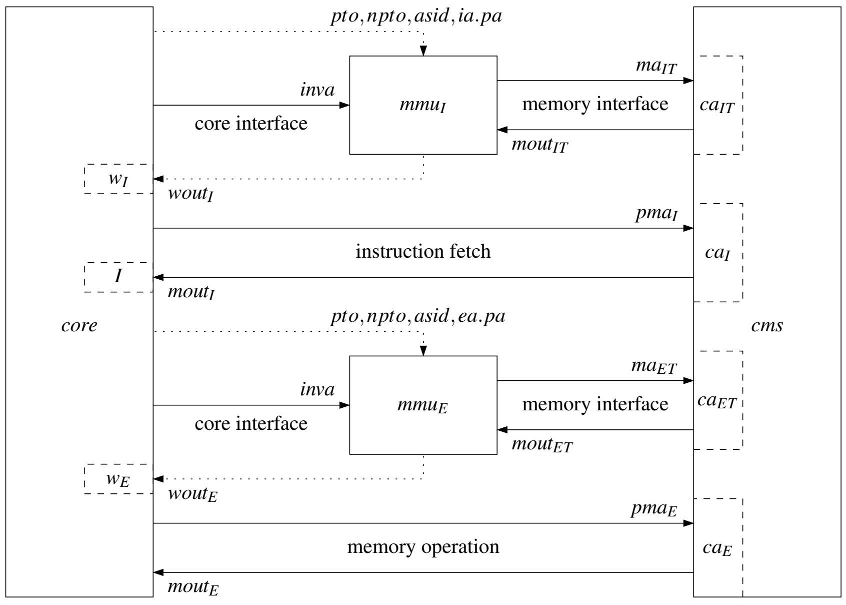


Fig. 28: Data paths connecting the MMUs to the processor core and the memory system

solution). Though the machine considered in this chapter is sequential, it uses the shared memory. This turns to be sufficient for hardware to perform some speculative in ISA translations (think, e.g., about the reset).

6.1.1 Control Logic

Execution of instructions in the sequential machine is subdivided into several phases. Within each phase, that can last several cycles, the hardware performs various actions in terms of execution of the next — with respect to the program order — instruction, which we refer to as the *current instruction* in what follows. Below we describe every phase by means of actions performed in the hardware.

i) instruction address translation. In case the machine runs in translated mode, instruction MMU (mmu_I) provides a translation for the current instruction.
ii) instruction fetch. Instruction cache (ca_I) provides the current instruction.
iii) effective address translation. In case the machine runs in translated mode and the current instruction is a memory operation, data MMU (mmu_E) provides a translation for the effective address.
iv) instruction execution. In case the current instruction is a memory operation, data cache (ca_E) provides the data required to perform this operation. Data structures of the sequential machine are updated by execution of the current instruction.

On interrupts, the hardware either jumps to the interrupt service routine or — in case the resume type is continue — continues its current operation. Control logic that implements this is not particularly difficult. The simple automaton in Fig. 29 is designed to keep track of the current execution phase (in one of the control states) and change the machine's control signals according to the execution phase, availability of the requested data, and the presence of unmasked interrupts. For convenience we introduce abbreviations for the following hardware signals.

$$\begin{aligned} exec(h) &\equiv jisr(h) \rightarrow cont(h) \\ mexec(h) &\equiv exec(h) \wedge mop(h) \end{aligned}$$

The first signal is used to specify the transitions of in the control automaton. Intuitively, execution of the current instruction continues while the latter signals is active. On the non-continue

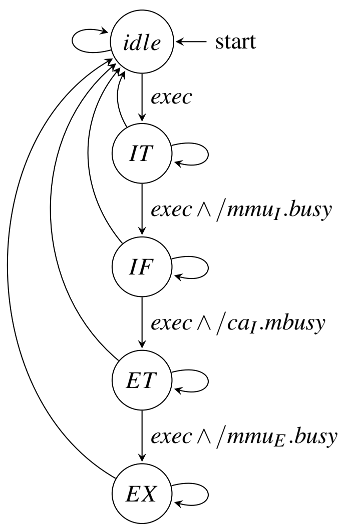

idle
start
exec
IT
$exec \wedge /mmu_I.busy$
IF
$exec \wedge /ca_I.mbusy$
ET
$exec \wedge /mmu_E.busy$
EX

Fig. 29: Machine's control automaton

type interrupts, execution of the current instruction is interrupted and a jump to the interrupt service routine is performed. For that purpose we added several transitions returning control to the initial state (see Fig. 29). States

$$\mathcal{S} = \{idle, IT, IF, ET, EX\}$$

of the control automaton we naturally order as follows:

$$idle < IT < IF < ET < EX.$$

To keep track of the current *control state* we introduce a function

$$cs : K_{HW} \to \mathcal{S}$$

which maps hardware configurations $h \in K_{HW}$ to control states $\sigma \in \mathcal{S}$ such that

$$\sigma(h) \; \to \; cs(h) = \sigma.$$

While a cache or an MMU stays busy, we say that the corresponding control state is *busy*, and abbreviate

$$busy(\sigma, h) \; \equiv \; \begin{cases} h.mmu_Y.busy & \sigma = YT \\ h.ca_I.busy & \sigma = IF \\ h.ca_E.busy & \sigma = EX \\ 0 & \sigma = idle. \end{cases}$$

From the construction of the control automaton we clearly have the following.

Lemma 46. *Assume $cs(h) \neq idle$.*

$$busy(cs(h), h) \; \leftrightarrow \; cs(h') = cs(h)$$

Control state $\sigma \in \mathcal{S}$ *finishes* if the control leaves state σ:

$$fin(\sigma, h) \; \equiv \; (cs(h) = \sigma) \wedge /busy(\sigma, h).$$

Execution of the current instruction ends either on an interrupt, or if state *EX* finishes:

$$endex(h) \; \equiv \; exec(h) \to fin(EX, h).$$

A trivial lemma follows directly from the construction of the control automaton.

Lemma 47.

$$endex(h) \rightarrow idle(h') \quad (1)$$
$$/endex(h) \rightarrow cs(h) \leq cs(h') \quad (2)$$

For convenience, we also abbreviate

$$finex(\sigma, h) = fin(\sigma, h) \wedge exec(h),$$

and another trivial lemma follows.

Lemma 48. *Assume $cs(h) \neq EX$.*

$$finex(cs(h), h) \;\rightarrow\; cs(h) < cs(h')$$

6.1.2 Collecting Interrupts

The event signals are collected in every control state visited throughout execution of the current instruction. For convenience we introduce the following shorthands to group the event signals collected in the various states.

$$\begin{aligned}
ev(idle) &= 0_9 \circ e \circ reset \\
ev(IT) &= 0_8 \circ malf \circ 0_2 \\
ev(IF) &= 0_6 \circ gff \circ pff \circ 0_3 \\
ev(ET) &= 0_2 \circ malm \circ ovf \circ sysc \circ ill \circ 0_5 \\
ev(EX) &= gfm \circ pfm \circ 0_9
\end{aligned}$$

Note that we mask the external interrupt signal outside the *idle* state.

$$e(h) = idle(h) \wedge eev(h)[1]$$

In case an external interrupt occurs in a cycle when the machine does not reside in the *idle* state, the latter interrupt is masked until the hardware returns into the *idle* state. The interrupt convention [MP00] makes sure that the external interrupt stays on, and eventually will be received by the hardware. Computation of most internal interrupt event signals precisely follows the specifications from Sect. 2.3, and therefore is omitted. For the page faults on fetch and memory operation resp. we have the following.

$$\begin{aligned}
pff(h) &= /host(h) \wedge f(h.w_I) \\
pfm(h) &= /host(h) \wedge f(h.w_E) \wedge mop(h)
\end{aligned}$$

Analogously, the general-protection faults on fetch and memory operation resp. are computed as follows:

$$\begin{aligned}
gff(h) &= /host(h) \wedge /f(h.w_I) \wedge (11 \not\leq h.w_I.r[\texttt{xu}]) \\
gfm(h) &= /host(h) \wedge /f(h.w_E) \wedge (1\texttt{s} \not\leq h.w_E.r[\texttt{uw}]) \wedge mop(h)
\end{aligned}$$

where write-bit s is computed according to the specification (see Sect. 3.3.5).

$$\texttt{s} \;\equiv\; s(h) \vee cas(h)$$

For the masked cause signal we get the following:

$$mca(h) = imask(h) \wedge \bigvee_{\sigma \leq cs(h)} ev(\sigma, h)$$

where the interrupt mask above is obtained from the status register.

$$imask(h) = 1^9 \circ h.sr[1] \circ 1.$$

Computation of processor control signals *jisr* and *cont* remains straightforward.

$$\begin{aligned} jisr(h) &\equiv mca(h) \neq 0_{11} \\ cont(h) &\equiv f1(mca(h))[7:6] \neq 0_2 \end{aligned}$$

In general, on JISR we have to argue only about those interrupt signals which are *used* in a particular control state of the machine. In order to keep track of the used interrupt signals we throw in another technical definition. When given a control state, auxiliary function

$$\mathcal{J} : \mathcal{S} \to \mathbb{N}^*$$

returns the set of interrupts (indices) processed in that state. To define $\mathcal{J}$ formally we simply specify all the values.

σ	*idle*	*IT*	*IF*	*IT*	*EX*
$\mathcal{J}(\sigma)$	$\{1\}$	$\{2\}$	$[3:4]$	$[5:8]$	$[9:10]$

A trivial property follows directly from the definition.

$$\forall \sigma_1, \sigma_2 \in \mathcal{S} : \; \sigma_1 \leq \sigma_2 \leftrightarrow \max \mathcal{J}(\sigma_1) \leq \max \mathcal{J}(\sigma_2)$$

Using the latter function we give an alternative definition for the *jisr* signal:

$$jisr(h) = \bigvee_{k \leq \max \mathcal{J}(cs(h))} mca(h)[k].$$

Below we introduce an invariant restricting the bits set in the masked cause signal based on the current control state.

Invariant 5.

$$\begin{aligned} IF \leq cs(h) \;&\to\; mca(h)[2:0] = 0_3 && (1) \\ mop(h) \wedge EX \leq cs(h) \;&\to\; mca(h)[8:0] = 0_9 && (2) \end{aligned}$$

Proof of invariant 5.1. By induction on the number of hardware cycles t. For the base case $(t = 0)$ there is nothing to show.

$$IF \not\leq cs(0) = idle$$

For the induction step from t to $t+1$ we split cases on the current control state $(cs(t) \in \mathcal{S})$. Moreover, we assume

$$exec(t) \wedge /fin(EX,t)$$

since otherwise we have $endex(t)$ and by lemma 47 there is nothing to show.

$$IF \not\leq cs(t+1) = idle$$

- $IF \leq cs(t)$. From the construction we have

  $$mca(t+1)[1:0] = 0_2.$$

  For the misalignment on fetch interrupts we derive

  $$\begin{aligned} mca(t+1)[2] &= malf(t+1) && \text{(definition)} \\ &= malf(t) && \text{(construction)} \\ &= mca(t)[2] && \text{(definition)} \end{aligned}$$

  and the claim follows from the induction hypothesis.

- $IF \not\leq cs(t)$. In this case we consider only

  $$IT(t) \wedge IF(t+1)$$

  since otherwise there is nothing to show. From the assumptions we derive

  $$\begin{aligned} exec(t) &\leftrightarrow /jist(t) \vee cont(t) && \text{(definition)} \\ &\leftrightarrow \forall k \leq \max \mathcal{J}(IT) : /mca(t)[k] && \text{(definition)} \\ &\leftrightarrow mca(t)[2:0] = 0_3 && \text{(definition)} \end{aligned}$$

  and the claim follows similarly as in the case above. □

Proof of the second part of invariant 5 is analogous to the proof of the first part, and therefore is omitted.

6.1.3 Implementation Registers

According to Fig. 28, implementation of the processor core of the sequential machine utilizes the following implementation (invisible) registers:

- $h.core.w_I, h.core.w_E \in \mathbb{B}^{60}$ — two walk registers resp. for fetching instructions and accessing the memory, and
- $h.core.I \in \mathbb{B}^{32}$ — an ordinary instruction register.

Obviously, the implementation registers are connected to the walk outputs of the corresponding MMUs and updated whenever the corresponding translation phase finishes.

$$\begin{aligned} h.w_Y.in &= h.mmu_Y.wout \\ h.w_Y.ce &= YT(h) \wedge /h.mmu_Y.busy \end{aligned}$$

The instruction register, as usual, is connected to the data output of the instruction cache and updated whenever the instruction fetch phase finishes. As simple as that.

$$\begin{aligned} h.I.in &= \begin{cases} h.ca_I.pdout_H & h.ca_I.pa[2] \\ h.ca_I.pdout_L & \text{otherwise} \end{cases} \\ h.I.ce &= IF(h) \wedge /h.ca_I.mbusy \end{aligned}$$

Basically, the walk registers are introduced to have realistic cycle times in hardware, whereas the instruction registers is necessary to support the self-modifying code. Without the instruction register we would have a problem with instructions writing at its own physical memory addresses.[2] Another trivial lemma follows directly from the construction of the control automaton and definitions of the update enable signals.

Lemma 49. *For states $\sigma \neq idle$ and registers Z we have:*

$$\sigma(h) \wedge \sigma(h') \ \rightarrow \ /h.core.Z.ce$$

[2] Without the instruction register, the instruction data would be coming straight from the instruction cache. Moreover, the instruction data — cache line containing the instruction — has to be present in the data cache in order to be modified. Using terminology from [KMP14], modification of the instruction data in the data cache would be a *global write* access, whereas fetching the instruction word from the data cache — a *local read* access. According to [KMP14], global accesses do not overlap with local reads except for the starting cycles.

In a design without the instruction register this would lead to a busy instruction cache in all cycles of the global access except for the starting one. The latter includes the cycle in which the data cache lowers the busy signal — in which the processor core is updated and the machine switches to the *idle* state. Thus, the processor request to a busy instruction cache is lowered, which violates the operating conditions of the cache memory system designed in [KMP14] (see Sect. 6.2.2).

6.1.4 Connecting Components

Here we collect in a bookkeeping manner the definitions which connect all components of the machine together. Some of these definitions, such as the ones that provide the MMUs with information about location of the page tables (*pto*, *npto*), are general in a sense that they apply to other machine types without changes.

Translation Queries

We raise translation request $treq_I$ (for the instruction address) if the machine runs in translated mode. We raise translation request $treq_E$ (for the effective address) in the translation mode as well but only for the memory operations.

$$\begin{aligned} treq_I(h) &\equiv /host(h) \wedge IT(h) \wedge exec(h) \\ treq_E(h) &\equiv /host(h) \wedge ET(h) \wedge mexec(h) \end{aligned}$$

Having that we formally define the translation request input of mmu_Y as follows.

$$h.mmu_Y.treq = treq_Y(h)$$

The dedicated special purpose registers provide to mmu_Y the memory locations of the guest and user page tables resp. as given below. For convenience we abbreviate

$$\begin{aligned} pto(h) &= h.pto.pa \\ npto(h) &= h.npto.pa \end{aligned}$$

and define:

$$\begin{aligned} h.mmu_Y.pto_G &= pto(h) \\ h.mmu_Y.pto_U &= npto(h). \end{aligned}$$

These inputs as well as the ASID portions of the translation address inputs (below) are invariant w.r.t. the MMU implementations. The reason being that any other definition would contradict the specification in part of the TLB steps (see Sect. 3.3.1).

The remaining part of the translation queries — translation address inputs — are defined for the two MMUs individually. Into mmu_I we plug the page address of the actual instruction address ($ia(h)$), obviously, prepended with the current ASID. Respectively, into mmu_E — the page address of the actual effective address ($ea(h)$), prepended in the same way.

$$\begin{aligned} h.mmu_I.upa &= asid(h) \circ ia(h).pa \\ h.mmu_E.upa &= asid(h) \circ ea(h).pa \end{aligned}$$

Invalidation Queries

We formalize the invalidation queries in the same way as the translation queries above: it is the data coming to an MMU through the invalidation inputs if (!) one of the invalidation request signals is high. We define the invalidation request inputs of mmu_Y using the machine control signals introduced above as follows.

$$\begin{aligned} h.mmu_Y.invlpg &= finex(EX,h) \wedge invlpg(h) \wedge \overline{user(h)} \\ h.mmu_Y.vmflush &= finex(EX,h) \wedge flusht(h) \wedge guest(h) \\ h.mmu_Y.flush &= finex(EX,h) \wedge flusht(h) \wedge host(h) \end{aligned}$$

As one can see, the same invalidation requests are raised for both MMUs. Universal page addresses which are the subject of these requests are formally defined below.

$$\begin{aligned} h.mmu_Y.inva.as &= \begin{cases} vmid(h) \circ A(h)[27:20] & guest(h) \\ A(h)[31:20] & \text{otherwise} \end{cases} \\ h.mmu_Y.inva.pa &= B(h)[31:12] \end{aligned}$$

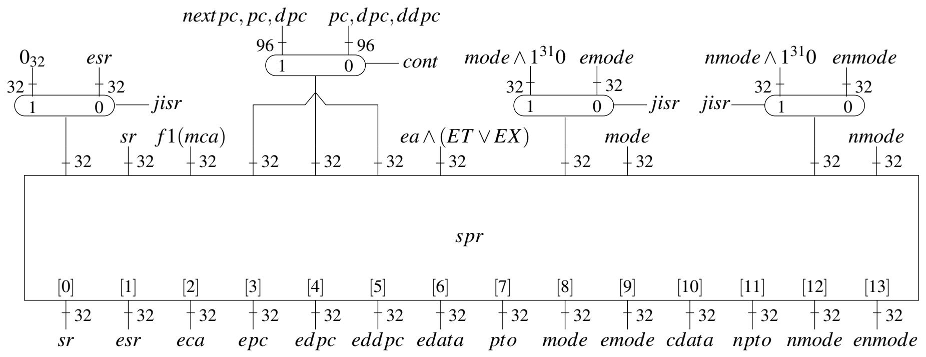


Fig. 30: Wiring of the special inputs of the SPR

SPR Environment

Wiring of the special purpose registers does not change much. We only need to

i) modify the input signals for registers *h.eca* and *h.edata*,
ii) modify the special input and clock enable signal for register *h.mode*, and
iii) provide the special input and clock enable signal for register *h.nmode*.

For registers *h.pto* and *h.npto* we do not change anything: still write them both manually, still via the *move* instructions. Register *h.enmode* is wired as an ordinary exception register, i.e., to backup the value of *h.nmode* on *jisr*.

Connection of the special input signals is depicted in Fig. 30. For the special purpose registers we follow the specifications from Sect. 3.3.7, which are pretty straightforward. Thus, the exception cause and the exception data registers are connected resp. to the (hardware) interrupt level and the effective address masked outside control states *ET* and *EX*.

$$\begin{aligned} h.eca.in &= 0_{21} \circ f1(mca(h)) \\ h.edata.in &= ea(h) \wedge (ET \vee EX)(h) \end{aligned}$$

For both mode registers

$$xmode \in \{mode, nmode\}$$

we connect the inputs as follows.

$$h.xmode.in = \begin{cases} h.xmode \wedge 1^{31}0 & jisr(h) \\ h.exmode & \text{otherwise} \end{cases}$$

In this way the least significant bit of the mode register is cleared on *jisr*, whereas all bits are restored from the corresponding exception register on *eret*. The special inputs are written into the mode registers under control of the special clock enable signals raised in accordance with the specification.

$$\begin{aligned} h.mode.ce &= jisr(h) \wedge (\overline{user(h)} \vee icpt(h)) \ \vee \ eret(h) \wedge host(h) \\ h.nmode.ce &= jisr(h) \wedge (user(h) \wedge \overline{icpt(h)}) \ \vee \ eret(h) \wedge \overline{host(h)} \end{aligned}$$

6.1.5 Execution Levels and Intercepts

Before we proceed, for every level of privilege we first argue about what instructions are possible/allowed (do not cause the illegal interrupt), and clearly we focus on execution of the translation and invalidation queries.

Level of Host

At the level of host the address translation is completely disabled. There are no translation queries to the MMUs since no translation requests are raised at this level (see Sect. 6.1.4).

$$host(h) \;\rightarrow\; /treq_I(h) \wedge /treq_E(h) \tag{7}$$

Thus, regular instructions (not invalidating the TLB cache) are executed in exactly the same way as in the sequential machine *without* address translation [PBLS16].

At the same time, instructions which invalidate translations act only on the TLB component and essentially are noops for the processor core. Both invalidating instructions are allowed at the level of host.

$$host(h) \wedge (invlpg(h) \vee flusht(h)) \;\rightarrow\; /ill(h) \tag{8}$$

Next we consider executions at the levels of guest and user in a similar way.

Level of Guest

At this (intermediate) level the address translation is enabled partially: only queries which request translations for addresses of the running guest are allowed.

$$guest(h) \wedge treq_Y(h) \;\rightarrow\; upa_Y(h) \in A_G(vmid(h)) \tag{9}$$

Analyzing the control logic of the nested MMU one easily derives from the latter restriction that the address translation at the level of guest is performed exclusively by the *simple MMU*. Portions of this component which implement the translation scheme — the control automaton and the walking unit — were incorporated from the machine *with* the ordinary address translation [LOP].

Both invalidating instructions are allowed at this level, but — in contrast to the level of host — the *invlpg* instruction is restricted to invalidate only translations that were accumulated by the running guest.

$$guest(h) \wedge (invlpg(h) \vee flusht(h)) \;\rightarrow\; /ill(h) \wedge inva_Y(h) \in A_G(vmid(h)) \tag{10}$$

Note, at the level of guest the *flusht* instruction is interpreted as the *vmflush* (see Sect. 3.3.6).

Level of User

At the level of user the address translation is enabled for both the user and the guest addresses. At this level the translation queries are restricted — by the specification — to addresses of the running user and addresses of the guest under which the user is running.

$$user(h) \wedge treq_Y(h) \;\rightarrow\; upa_Y(h) \in A_U(vmid(h), prid(h)) \tag{11}$$

For the guest addresses everything is simple: translation is performed exactly as in the case above (for the level of guest), utilizing exactly the same hardware components. The translation queries for the guest addresses arise in the process of the nested translation — translation of the user addresses. This process was semi-formally described in Sect. 3.2.1, formally specified — as an iterative process with restricted steps — in Sect. 3.3.1, and implemented in hardware in Sect. 4.2.2. The fact that our hardware implementation does not violate the restrictions imposed by the specification we formally proved in Sect. 5.2.

At the level of user invalidating instructions are forbidden in any form.

$$user(h) \wedge (invlpg(h) \vee flusht(h)) \;\rightarrow\; ill(h) \tag{12}$$

Triggering Intercepts

Finally, we proceed to specify computation of the intercept signal. According to the specification in Sect. 3.3.5, the intercepts are triggered only at the level of user. Recall, these intercepts were introduced in order to make the page faults that occur in the process of nested translation (due to translations of intermediate guest addresses) completely invisible at the level of guest. Thus, given that the machine resides in control stage

$$\sigma \in \{IF, ET, EX\}$$

we activate the intercept signal in case translation $h.w_I$ for the instruction address indicates a guest level fault. Analogously, given that the machine resides in control state

$$\sigma \in \{EX\}$$

and a memory operation is performed, we activate the intercept signal if translation $h.w_E$ for the effective address indicates a guest level fault. Combining the latter two causes, we clearly obtain the following.

$$\begin{aligned} icpt(h) \;=\; & user(h) \wedge (IF \le cs(h) \wedge h.w_I.f_g \;\vee \\ & EX \le cs(h) \wedge h.w_E.f_g \wedge mop(h)) \end{aligned}$$

This finishes construction of our sequential processor. In the next section we interconnect it with a cache memory system, which will complete the sequential implementation.

6.2 Cache Memory System in Sequential Processor

In this chapter we consider a single-core sequential processor (as constructed above) connected to a sequentially consistent cache memory system (as constructed in [KMP14]) with four caches: two caches for translation of the instruction and effective addresses and two resp. for the instruction fetch and the data access.

6.2.1 Connections to Caches

There are four caches available to the processor core and two MMUs for accesses.[3]

$$\begin{aligned} h.ca_Y &= h.ca(2\mathbb{1}_{\{Y=E\}} + 1) \\ h.ca_{YT} &= h.ca(2\mathbb{1}_{\{Y=E\}}) \end{aligned}$$

We raise the processor requests to these caches as in [LOP]: the processor request to cache ca_Y is, obviously, raised by the processor core, while the request to cache ca_{YT} is raised by mmu_Y. The requests to caches ca_I and ca_E are defined in the obvious way.

$$\begin{aligned} h.ca_I.preq &= IF(h) \wedge exec(h) \\ h.ca_E.preq &= EX(h) \wedge mexec(h) \\ h.ca_{YT}.preq &= h.mmu_Y.mreq \end{aligned}$$

Since in the scope of this chapter the address translation remains our main interest, we proceed to specify the processor addresses for the memory accesses. To define the processor address at port Y, we introduce the *physical memory addresses* (pma_Y):

[3] Note, the indicator function used above is defined for $a \in \mathbb{B}$ simply as follows:

$$\mathbb{1}_{\{a\}} = \begin{cases} 1 & a \\ 0 & \overline{a}. \end{cases}$$

$$
\begin{aligned}
pma_I(h) &= pma_I(h).pa \circ ia(h).po \\
pma_E(h) &= pma_E(h).pa \circ ea(h).po
\end{aligned}
$$

where the page address of the physical memory address at port Y is

$$
pma_Y(h).pa = \begin{cases} h.mmu_Y.pa & host(h) \\ h.mmu_Y.wout.ba & \text{otherwise.} \end{cases}
$$

Thus, the physical memory address for port I is either the output of mmu_I followed by the page offset of $ia(h)$ or simply $ia(h)$, depending on whether the machine runs in the translated mode or not resp. Transferring this onto port E: pma_E is either the output of mmu_E followed by the page offset of $ea(h)$ or simply $ea(h)$. The processor address provided to cache ca_{YT} is coming from mmu_Y (see Fig. 28).

$$
\begin{aligned}
h.ca_Y.pa &= pma_Y(h).l \\
h.ca_{YT}.pa &= h.mmu_Y.ma.l
\end{aligned}
$$

The remaining parts of the memory accesses are obvious and, moreover, the same as defined in [LOP]. Thus, for the access types we simply have the following.

$$
\begin{aligned}
h.ca_I.(pr, pw, pcas) &= 100 \\
h.ca_E.(pr, pw, pcas) &= l(h) \circ s(h) \circ cas(h) \\
h.ca_{YT}.(pr, pw, pcas) &= 100
\end{aligned}
$$

Data for the memory accesses at port E are provided in accordance with the ISA.

$$
\begin{aligned}
h.ca_E.pbw &= bw(h) \\
h.ca_E.pdin &= dmin(h) \\
h.ca_E.pcdin &= cdata(h)
\end{aligned}
$$

6.2.2 Stability of Inputs to Caches

In order to satisfy the operating conditions of the cache memory system, we have for all ports to guarantee that until the processing of an ongoing memory access finishes all inputs of that access stay constant in the digital sense [LOP]. For the machine from [KMP14] this property was reflected in lemma 9.11 for the instruction and the data cache. Below we establish a counterpart of the latter lemma for our design with four caches.

Lemma 50.

- *For any cache:*
$$h^t.ca(i).mbusy \;\rightarrow\; h^t.ca(i).preq \tag{13}$$
- *For the translation caches:*
$$h^t.ca_{YT}.mbusy \;\rightarrow\; h^{t+1}.mmu_Y.(ma, mreq) = h^t.mmu_Y.(ma, mreq) \tag{14}$$
- *For the instruction and data caches:*
$$h^t.ca_Y.mbusy \;\rightarrow\; \forall Z : /h^t.core.Z.ce \tag{15}$$

Proof of lemma 50. For cache i we argue by induction on the number of hardware cycles t. For the base case ($t = 0$) there is nothing to show.

$$h^0.ca(i).mbusy = 0$$

For the induction step from t to $t+1$ we first show equations 14 and 15 using the induction hypothesis (equation 13).

From the construction of the nested MMU (see Sect. 4.2.2), for the instruction translation and the data translation cache we know that the processor address is taken directly from the internal walk registers, which are not updated in cycles when the memory is busy. The processor request signal from MMU is kept stable by the nested control automaton (see Sect. 4.2.2). For the instruction cache we argue as follows.

$$\begin{aligned} h^t.ca_I.mbusy &\rightarrow h^t.ca_I.preq \qquad \text{(equation 13)} \\ &\rightarrow IF(t) \wedge exec(t) \qquad \text{(definition)} && (16) \\ &\rightarrow IF(t) \wedge IF(t+1) \qquad \text{(interconnect)} && (17) \\ &\rightarrow \forall Z : /h^t.core.Z.ce \qquad \text{(lemma 49)} \end{aligned}$$

The corresponding arguments for the data cache are analogous, and therefore are omitted. To complete the induction step we proceed as follows. We assume

$$h^{t+1}.ca(i).mbusy = 1,$$

since otherwise there is nothing to show. Next we split cases of whether cache i is busy in cycle t.

- In case cache i is busy
$$h^t.ca(i).mbusy = 1,$$
the claim follows directly from the lines above. For the instruction translation cache we derive:
$$\begin{aligned} h^t.ca_{IT}.mbusy &\rightarrow h^t.ca_{IT}.preq \qquad \text{(equation 13)} \\ &\rightarrow h^t.mmu_I.mreq \qquad \text{(interconnect)} \\ &\rightarrow h^{t+1}.mmu_I.mreq \qquad \text{(equation 14)} \\ &\rightarrow h^{t+1}.ca_{IT}.preq \qquad \text{(interconnect).} \end{aligned}$$
In turn, for the instruction cache we simply have:
$$\begin{aligned} h^t.ca_I.mbusy &\rightarrow IF(t+1) \wedge exec(t) \qquad \text{(equations 17, 16)} \\ &\rightarrow IF(t+1) \wedge exec(t+1) \qquad \text{(equation 15)} \\ &\rightarrow h^{t+1}.ca_I.preq \qquad \text{(definition).} \end{aligned}$$
The corresponding arguments for the data translation cache and the data cache are analogous, and therefore are omitted.
- Otherwise, in case cache i is not busy
$$h^t.ca(i).mbusy = 0,$$
from the construction of caches [KMP14] we know that the *master automaton* of cache i resides and remains in its idle state.
$$idle(i)^t \wedge idle(i)^{t+1}$$
Thus, the only way to become busy in cycle $t+1$ is to receive a processor request
$$h^{t+1}.ca(i).preq = 1$$
which gives the claim. □

6.2.3 Accesses of Hardware Computation

Recall from [KMP14], the cache memory system used is accessed by accesses from the so-called multi-port access sequence

$$acc : [0:4p-1] \times \mathbb{N} \to K_{acc}$$

where p denotes the number of processors (in the scope of this chapter $p = 1$). Also recall that the latter sequence is defined with the help of auxiliary function $e(i,k)$, which gives the ending cycle of access (i,k). Thus, for non-flushing access k to the instruction cache we specify:

$$\begin{aligned} acc(1,k).a &= pma_I(e(1,k)).l \\ acc(1,k).type &= 1000. \end{aligned}$$

For non-flushing access k to the data cache ending in cycle $t = e(3,k)$ we have:

$$\begin{aligned} acc(3,k).a &= pma_E(t).l \\ acc(3,k).data &= dmin(t) \\ acc(3,k).cdata &= cdata(t) \\ acc(3,k).bw &= bw(t) \\ acc(3,k).type &= l(t) \circ s(t) \circ cas(t) \circ 0. \end{aligned}$$

Finally, for non-flushing access k to translation cache ca_{YT} ending in cycle

$$t = e(2\mathbb{1}_{\{Y=E\}},k)$$

we specify:

$$\begin{aligned} acc(2\mathbb{1}_{\{Y=E\}},k).a &= mmu_Y^t.ma.l \\ acc(2\mathbb{1}_{\{Y=E\}},k).type &= 1000. \end{aligned}$$

All accesses ending in cycle t are collected into the set

$$E(t) = \{(i,k) \mid e(i,k) = t\}.$$

Note, the set above also includes accesses generated by the cache memory system internally (flushes). For convenience we abbreviate the number of ending accesses by

$$ne(t) = \#E(t).$$

As before, the total number of accesses ending *before* cycle t is given by function NE.

$$\begin{aligned} NE(0) &= 0 \\ NE(t+1) &= NE(t) + ne(t) \end{aligned}$$

Since we use a sequentially consistent memory system (from [KMP14]), we know there is a sequential order

$$seq : \ [0:4p-1] \times \mathbb{N} \to \mathbb{N}$$

in which the accesses are performed. Therefore, every access performed in cycle t gets some sequential number

$$\forall a \in E(t)\ \exists n \in [0:ne(t)-1] : \ seq(a) = NE(t) + n$$

in the singe-port access sequence acc'. Recall, by definition of sequence acc' we have

$$acc'[seq(a)] = acc(a).$$

Using sequence acc' we proceed to formulate an important property of the memory abstraction. Namely, part 3 of lemma 8.65 in [KMP14], which we require later in the correctness proof.

Lemma 51 (relating hardware with atomic protocol).

$$m(h^t) = \Delta_M^{NE(t)}(m(h^0), acc')$$

In order to argue about the answers to read and CAS accesses, we would also require lemma 8.67 of [KMP14]. Again for convenience we repeat the statement.

Lemma 52 (sequential consistency). *Let $e(i,k) = t$ and $acc(i,k).r$ or $acc(i,k).cas$. Then*

$$pdout(i)^t = \Delta_M^{seq(i,k)}(m(h^0), acc')(acc(i,k).a).$$

6.2.4 Relating Endings of Accesses with Hardware Control Signals

In this section we repeat the arguments made in Sect. [9.3.5] of [KMP14] about accesses to the cache memory system performed by the processor. As before, now the accesses are also performed by the MMUs. In the first lemma below we reflect that every walk extension performed by the MMU is always accompanied by a read access ending in the cache utilized by that MMU.

Lemma 53.

$$wext(mmu_I^t) \rightarrow \exists k : e(0,k) = t \wedge acc(0,k).r \tag{1}$$

$$wext(mmu_E^t) \rightarrow \exists k : e(2,k) = t \wedge acc(2,k).r \tag{2}$$

In the second lemma we show the opposite: every read access ending in the cache utilized by the MMU triggers a walk extension step on that MMU.

Lemma 54.

$$e(0,k) = t \wedge acc(0,k).r \rightarrow (\ mmu_I^t.treq \rightarrow wext(mmu_I^t)\ ) \tag{1}$$

$$e(2,k) = t \wedge acc(2,k).r \rightarrow (\ mmu_E^t.treq \rightarrow wext(mmu_E^t)\ ) \tag{2}$$

Since the presented arguments remain trivial, we omit the proofs. Lemma 55 below is the counterpart of lemmas [9.12] and [9.13] resp. from [KMP14] about accesses performed by the processor core. The latter lemma is formulated to fit the sequential processor utilizing four caches. Thus, every time state *IF* finishes, a read access in the instruction cache ends and vice versa. The same holds for state *EX* and the data cache on execution of memory operations.

Lemma 55.

$$finex(IF,t) \leftrightarrow \exists k : e(1,k) = t \wedge acc(1,k).r \tag{1}$$

$$mop(t) \wedge finex(EX,t) \leftrightarrow \exists k : e(3,k) = t \wedge \overline{acc(3,k).f} \tag{2}$$

Proofs of the last three lemmas are straightforward, and therefore are omitted.

6.3 Liveness

In this section we show that the sequential machine constructed in Sects. 6.1 and 6.2 is live. In the end of Sect. 6.3.1 we show that every ISA instruction is eventually executed by the sequential processor (lemma 60). Then, in Sect. 6.3.2, we show the following: if a given control state is visited on execution of a given instruction, it is visited exactly once (equation 19).

6.3.1 Liveness of Control States

In the first lemma proven in this section we show that the control always eventually leaves the current control state.

Lemma 56.

$$busy(cs(t),t) \rightarrow \exists t' > t : /busy(cs(t),t')$$

Proof of lemma 56. By case split on the current control state ($cs(t) \in \mathcal{S}$):

- $cs(t) = idle$. For state *idle* there is nothing to show, since by definition we have

$$busy(idle,t) = 0.$$

- $cs(t) = IT$. In this case we argue using liveness of the nested MMU as follows:

$$\begin{aligned} busy(IT,t) &\leftrightarrow mmu_I^t.busy && \text{(definition)} \\ &\rightarrow \exists t' > t : /mmu_I^{t'}.busy && \text{(lemmas 29–31)} \\ &\leftrightarrow /busy(IF,t') && \text{(definition).} \end{aligned}$$

- $cs(t) = IF$. In state IF we rely on liveness of the instruction cache:

$$\begin{aligned} busy(IF,t) &\leftrightarrow h^t.ca_I.mbusy && \text{(definition)} \\ &\rightarrow h^t.ca_I.preq && \text{(lemma 50)} \\ &\rightarrow \exists t' > t : /h^t.ca_I.mbusy && \text{(lemma 8.68 of [KMP14])} \\ &\leftrightarrow /busy(IF,t') && \text{(definition).} \end{aligned}$$

- $cs(t) \in \{ET, EX\}$. The arguments for states ET and EX are analogous to those presented above for states IT and IF resp., and therefore are omitted. □

For convenience, we rewrite the latter result as follows.

Lemma 57.
$$/fin(cs(t),t) \;\rightarrow\; \exists t' > t : \; fin(cs(t),t')$$

Proof of lemma 57. Unfolding the definition we obtain:

$$\begin{aligned} /fin(cs(t),t) &\leftrightarrow (cs(t) \neq cs(t)) \vee busy(cs(t),t) \\ &\leftrightarrow busy(cs(t),t). \end{aligned}$$

In case *idle* is the current control state, there is nothing to show.

$$busy(idle,t) = 0$$

Otherwise, we argue as follows. From lemma 56, for some cycle $t' > t$ we have

$$busy(cs(t),t') = 0.$$

For minimum such cycle t' we clearly have

$$\forall \tilde{t} \in [t : t'-1] : \; busy(cs(t),\tilde{t}),$$

and therefore

$$\forall \tilde{t} \in [t : t'] : \; cs(\tilde{t}) = cs(t).$$ □

Using the lemma above we proceed to show the following: the control always eventually leaves every control state except EX, either to state *idle* (on core steps) or to the *next* control state. Note, the control leaves state EX only to state *idle* (see Fig. 29).

Lemma 58.
$$cs(t) \neq EX \;\rightarrow\; \exists t' \geq t : \; cstep(t') \vee cs(t') > cs(t)$$

Proof of lemma 58. From lemma 57, for some cycle $t' \geq t$ we have

$$fin(cs(t),t').$$

We assume $exec(t')$, since otherwise the claim immediately follows for cycle t'.

$$/exec(t') \;\rightarrow\; cstep(t')$$

Therefore, we have

$$finex(cs(t),t')$$

which by lemma 48 implies

$$cs(t'+1) > cs(t') = cs(t)$$

and the claim follows for cycle $t'' = t'+1$. □

From the latter lemma one easily concludes the following result.

$$cs(t) \neq EX \;\rightarrow\; \exists t' \geq t: \; cstep(t') \vee (cs(t') = EX) \tag{18}$$

Next we argue that the processor core is stepped infinitely often.

Lemma 59.

$$/cstep(t) \;\rightarrow\; \exists t' > t: \; cstep(t')$$

Proof of lemma 59. By case split on the current control state ($cs(t) \in \mathcal{S}$):

- $cs(t) = EX$. State EX is busy in cycle t, since otherwise we obtain a contradiction.

  $$fin(EX,t) \;\rightarrow\; cstep(t)$$

  From lemma 57, for some cycle $t' > t$ we have

  $$fin(EX,t') = 0$$

  and the claim follows.
- $cs(t) \neq EX$. Using equation 18, for some cycle $t' > t$ we conclude

  $$cstep(t') \vee (cs(t') = EX).$$

  For $cstep(t')$ there is nothing to show. Otherwise, for cycle t' we conclude

  $$/cstep(t') \wedge (cs(t') = EX)$$

  and the claim follows exactly as in the case above, for some cycle $t'' > t'$. □

Finally, we show that the sequential implementation is live, i.e., we show that every ISA instruction is eventually executed.

Lemma 60.

$$\forall i \; \exists t: \; i = I(t) \wedge cstep(t)$$

Proof of lemma 60. By induction on index i of the current instruction ($I(t)$). For the induction base ($i = 0$) recall that immediately after reset we have by definition

$$I(0) = 0.$$

From lemma 59 we know the processor core is stepped infinitely often. Consider the first such cycle $t' > 0$ in which the processor core performs a step. Clearly, we have

$$(\forall t \in [0 : t' - 1] : /cstep(t)) \wedge cstep(t').$$

For the scheduling function we conclude by definition

$$I(t') = I(0) = 0.$$

For the induction step from i to $i+1$ we argue as follows. From the induction hypothesis we know there is a cycle t such that

$$i = I(t) \wedge cstep(t).$$

By definition we immediately obtain

$$I(t+1) = i + 1.$$

Applying lemma 59 we know there is a cycle $t' \geq t+1$:

$$cstep(t) \wedge (\forall \tilde{t} \in [t+1 : t'-1] : /cstep(\tilde{t})) \wedge cstep(t').$$

To complete the induction step we argue using the definition:

$$I(t') = I(t+1) = i + 1.$$ □

6.3.2 Uniqueness of Finish Cycles

In Sect. 7.1.4 we must define the oracle inputs for the current instruction in cycles in which the current instruction progresses through the visited control states. Therefore, we first must show that the latter cycles are unique. This last section is spent to derive that result. Below we argue that i) only the current control state can be busy and ii) a busy state guarantees the *exec* signal is active.

Lemma 61. *For any control state $\sigma \in \mathcal{S}$:*

$$busy(\sigma,t) \ \rightarrow\ (cs(t)=\sigma)\wedge exec(t)$$

Proof of lemma 61. By induction on the number of hardware cycles t. For the base case ($t=0$) there is nothing to show.

$$busy(\sigma,0)=0$$

For the induction step from t to $t+1$ we argue as follows. In case state σ becomes busy in cycle $t+1$

$$busy(\sigma,t)=0,$$

we split cases on the value of $\sigma \in \mathcal{S}$:

- $\sigma = idle$. For state *idle* there is nothing to show, since by definition we have

$$busy(idle,t+1)=0.$$

- $\sigma = YT$. From the construction of the control automaton of the nested MMU (Sect. 4.2.2), we immediately conclude

$$h^{t+1}.mmu_Y.treq=1$$

  and the claim follows from the hardware interconnect.
- $\sigma \in \{IF, EX\}$. The claim follows analogous to the case above, from the first part of lemma 50 (equation 13).

Otherwise, if state σ remains busy in cycle t

$$busy(\sigma,t)=1,$$

from the induction hypothesis we have

$$cs(t)=\sigma\neq idle$$

and from lemma 46 we conclude

$$cs(t+1)=cs(t).$$

Moreover, using lemma 49 we obtain

$$h^t.core.Z.ce=0,$$

and therefore

$$mca(t+1)[10:2]=mca(t)[10:2].$$

From the construction we also have

$$mca(t+1)[1:0]=0_2=mca(t)[1:0].$$

Thus, we conclude

$$exec(t+1)=exec(t)$$

and the claim follows from the induction hypothesis. □

Using the result above we proceed to show the following: if a given control state is visited on execution of a given instruction, it necessarily finishes on execution of that instruction.

Lemma 62. *For states $\sigma \in \mathcal{S}$ the following holds:*

$$\forall i:\ (\forall t:\ i = I(t) \rightarrow \sigma \neq cs(t)) \vee (\exists t:\ i = I(t) \wedge fin(\sigma,t))$$

Proof of lemma 62. Equivalently, we proceed to show the following statement.

$$\exists t:\ I(t) = i \wedge cs(t) = \sigma\ \rightarrow\ \exists t':\ I(t') = i \wedge fin(\sigma,t')$$

From lemma 57, for some cycle $t' \geq t$ we have

$$fin(\sigma,t').$$

For the minimum such cycle t' we clearly have

$$\forall \tilde{t} \in [t : t'-1] : /fin(\sigma,\tilde{t})$$

and therefore

$$\forall \tilde{t} \in [t : t']:\ cs(\tilde{t}) = cs(t) = \sigma.$$

Moreover, from the definition of finish cycles we derive

$$\begin{aligned} /fin(\sigma,\tilde{t}) &\leftrightarrow (cs(\tilde{t}) \neq \sigma) \vee busy(\sigma,\tilde{t}) \\ &\leftrightarrow busy(\sigma,\tilde{t}) \end{aligned}$$

which by lemma 61 implies

$$\forall \tilde{t} \in [t : t'-1] : exec(\tilde{t})$$

and therefore

$$\forall \tilde{t} \in [t : t'-1] : /cstep(\tilde{t}).$$

Therefore, for the scheduling function we conclude

$$\forall \tilde{t} \in [t : t']:\ I(\tilde{t}) = I(t) = i$$

which completes the proof. □

Finally, we strengthen the result above as follows: if a given control state is visited on execution of a given instruction, it finishes exactly once on execution of that instruction.

$$\forall i\, \forall \sigma:\ (\forall t:\ i = I(t) \rightarrow \sigma \neq cs(t)) \vee (\exists!\, t:\ i = I(t) \wedge fin(\sigma,t)) \tag{19}$$

Proof of equation 19. From lemma 62 for states $\sigma \in \mathcal{S}$ we know

$$\exists t:\ I(t) = i \wedge cs(t) = \sigma\ \rightarrow\ \exists t':\ I(t') = i \wedge fin(\sigma,t').$$

Assume state $\sigma \in \mathcal{S}$ finishes in cycles t and $t' > t$

$$fin(\sigma,t) \wedge fin(\sigma,t')$$

with instruction

$$I(t) = I(t').$$

Next we split cases on the current control state (in cycle t):

- $\sigma = EX$. In state EX we conclude $cstep(t)$, and derive a contradiction.

$$\begin{aligned} I(t') &\geq I(t+1) && \text{(monotonicity)} \\ &= I(t)+1 && \text{(definition)} \\ &= I(t')+1 && \text{(assumption)} \end{aligned}$$

- $\sigma \neq EX$. In this case we assume $/cstep(t)$, since otherwise the claim follows as above. From the construction of the control automaton (from lemma 48), we clearly have

  $$cs(t+1) > cs(t) = cs(t')$$

  which implies

  $$\exists \tilde{t} \in [t+1 : t'-1] : \; cstep(\tilde{t})$$

  since otherwise, again by construction of the control automaton (by lemma 47), we obtain a contradiction.

  $$cs(t+1) \leq cs(t')$$

  Still, the contradiction follows exactly as in the case above.

  $$\begin{aligned} I(t') &\geq I(\tilde{t}+1) && \text{(monotonicity)} \\ &= I(\tilde{t})+1 && \text{(definition)} \\ &\geq I(t)+1 && \text{(monotonicity)} \\ &= I(t')+1 && \text{(assumption)} \end{aligned}$$

  □

The latter uniqueness result completes the implementation of the sequential machine. Correctness of the sequential implementation is proven in the next chapter.

7

Correctness of Sequential Implementation

In order to prove that the hardware construction of the sequential machine with NAT from Chap. 6 is correct, i.e., it implements the ISA from Sect. 3.3, in Sect. 7.1 we state a simple (simulation) theorem. After elaborating of some more auxiliary machinery in Sect. 7.2, we proceed to show most of the correctness arguments constituting the induction step (correctness for registers) in Sect. 7.3. In Sect. 7.4 we show that our machine construction fulfills the correctness conditions on the MMU interfaces formulated in Sect. 3.4. This allows us to extend results on MMU hardware correctness established for the general semantics in Chap 5 to a 'proper' simulation relation (in the original semantics). Finally, in Sect. 7.5 we complete the induction step by showing that the guard conditions are obeyed in the computations performed by the sequential implementation.

7.1 Correctness Statement

Below we structure the arguments as follows. In Sect. 7.1.1 in the obvious way we specify the hardware configurations in which steps of the ISA computation are generated. This allows us to formulate the simulation theorem in Sect. 7.1.3. In Sect. 7.1.4 the definition of the stepping function is completed. Note, the software conditions for the sequential implementation are discussed in Sect. 7.1.2.

7.1.1 Stepping of Components

In this small section we cover definitions that concern the specification steps produced by the sequential machine. Basically the stepping stays the same as it was defined in [LOP]. Thus, the processor core step is tied to the end of the instruction execution.

$$cstep(h) \; = \; endex(h)$$

Note, a processor core step is performed on two occasions: i) execution of the current instruction uninterrupted (*exec*) and ii) jump to the interrupt service routine due to an unmasked interrupt of a non-continue type ($/exec$). In the first case the machine has to reside in control state *EX* and the data cache must not be busy.

Recall, the definition of *tadd* from Sect. 5.2 which specifies the total number of translation steps made in the given hardware configuration for the given MMU. For mmu_Y we introduce

$$tstep_Y(h) \; = \; tadd(h.mmu_Y)$$

and instantiate the *number of steps* function for our sequential machine.

$$ns(h) = tstep_I(h) + tstep_E(h) + cstep(h)$$

Analyzing the definition above we argue that the total number of steps performed by the sequential machine in the given hardware configuration is at most one.

Lemma 63.

$$ns(h) \leq 1$$

In order to define the stepping function — specify the steps performed in every hardware configuration — we first need to enumerate the steps globally. In turn this global ordering is defined recursively on the number of hardware cycles t. In all predicates and functions p above we switch from "current" configurations h to configurations h^t (configurations h in hardware cycle t), which for convenience we abbreviate:

$$p(t) \equiv p(h^t).$$

Now the "global" *number of steps* function ($NS(t)$) is composed as usual: from the numbers of steps performed in every single hardware cycle ($ns(t)$).

$$\begin{aligned} NS(0) &= 0 \\ NS(t+1) &= NS(t) + ns(t) \end{aligned}$$

7.1.2 Software Conditions

From the software conditions listed in Sect. 2.2.2 remains only the one forbidding store or CAS operations on ROM. Below we formulate the latter condition to fit the machine description.

$$SC(i) \equiv write^i_\sigma \to \langle pma^i_{E\ \sigma}.l \rangle \geq 2^r \tag{20}$$

The other software condition forbidding the misaligned memory accesses is discarded. Alignment of the memory accesses is tested by hardware, and if violated, one of the misalignment interrupts is raised.

7.1.3 Simulation Theorem

We present the result as usual — in the form of a simulation theorem. This section is used to state the latter theorem. The statement of the theorem repeats the corresponding statement for the sequential machine *without* the address translation. Formally we claim: *there exists an initial ISA configuration c^0 s.t. for all hardware cycles t the simulation relation — as defined later in this section — between configurations h^t of hardware and $c^{NS(t)}$ of ISA holds and the guard conditions — as defined later in this section — are satisfied for all steps before $NS(t)$:*

$$\exists c^0\ \forall t : sim(h^t, c^{NS(t)}) \wedge \Gamma^{NS(t)}$$

where

$$\begin{aligned} h^{t+1} &= \delta_{HW}(h^t) \\ c^{n+1} &= \delta_{ISA}(c^n, s(n)). \end{aligned}$$

Simulation Relation

Below we formulate the simulation relation for the sequential machine with nested MMUs. It is quite obvious to define: we say that the simulation relation (*sim*) for the entire machine holds iff

- both hardware TLBs, for instruction addresses and for the effective address, are simulated by the software TLB, and
- the hardware core and the cache memory system are simulated resp. by the specification core and the specification memory.

Formally, for the hardware configuration h and the ISA configuration c we define

$$sim(h,c) \equiv sim_{p+m}(h,c) \wedge sim_{tlb\times 2}(h,c)$$

where the individual simulation relations prescribe the simulation of the processor and memory (sim_{p+m}) and the TLBs ($sim_{tlb\times 2}$) resp. The former one is taken literally from [LOP]; we provide the definition for completeness of presentation.

$$\begin{aligned} sim_{p+m}(h,c) \equiv\ & c.(pc,dpc,ddpc) = h.(pc,dpc,ddpc) \wedge \\ & c.(gpr,spr) = h.(gpr,spr) \wedge \\ & \ell(c.m) = m(h) \end{aligned}$$

The latter one obviously comprises two parts: one for simulation of the TLB for instruction addresses, and one for simulation of the TLB for effective addresses.

$$sim_{tlb\times 2}(h,c) \equiv sim_{tlb}(h.mmu_I,c) \wedge sim_{tlb}(h.mmu_E,c)$$

Recall, simulation relation sim_{tlb} between the TLB of hardware configuration $h.mmu$ and the TLB of ISA configuration c was defined in section Sect. 5.2.2. Note that we formulated the latter relation in Chap. 5 to show correctness of the nested MMU implementation w.r.t. the general semantics, defined in Sect. 3.4. In this chapter we use this relation unchanged to show correctness of the nested MMU implementation w.r.t. the ordinary semantics.

Guard Conditions

In the obvious way we formalize the guard conditions for the sequential machine with NAT. Using the machinery from Sect. 3.3, we collect into a single predicate Γ all guard conditions for the computational step performed in ISA configuration c under oracle input $\sigma \in \Sigma$:

$$\Gamma(c,\sigma) \equiv \begin{cases} \Phi(c,\sigma) & \sigma \in \Sigma_{core} \\ T(c,\sigma) & \sigma \in \Sigma_{tlb}. \end{cases}$$

In case the guard conditions hold for all ISA steps before global step n, we define

$$\Gamma^n \equiv \forall m < n : \Gamma(c^m, s(m)).$$

7.1.4 Stepping Inputs

The stepping function is defined for every global step number ($NS(t)$) by a case split — on the type of step — as follows.

- In case a translation step is performed and the TLB for instruction addresses initiates this step, we essentially use the definition from [LOP], very slightly adjusted to specifics of the sequential machine.

$$tstep_{I\ G}(t) \rightarrow s(NS(t)) = \begin{cases} (winit, upa_G(mmu_I^t).pa, guest) & winit_G(mmu_I^t) \\ (wext, mmu_I^t.w_G, \bot) & \text{otherwise} \end{cases}$$

$$tstep_{I\ U}(t) \rightarrow s(NS(t)) = \begin{cases} (winit, upa_U(mmu_I^t).pa, user) & winit_U(mmu_I^t) \\ (wext, mmu_I^t.w_U, mmu_I^t.w_G) & \text{otherwise} \end{cases}$$

- If a translation step is performed and the TLB for effective addresses initiates this step, we simply follow the same intuition as in the case above.

$$tstep_{E\ G}(t) \rightarrow s(NS(t)) = \begin{cases} (winit, upa_G(mmu_E^t).pa, guest) & winit_G(mmu_E^t) \\ (wext, mmu_E^t.w_G, \bot) & \text{otherwise} \end{cases}$$

$$tstep_{E\ U}(t) \rightarrow s(NS(t)) = \begin{cases} (winit, upa_U(mmu_E^t).pa, user) & winit_U(mmu_E^t) \\ (wext, mmu_E^t.w_U, mmu_E^t.w_G) & \text{otherwise} \end{cases}$$

- For processor core steps ($cstep(NS(t))$) we assemble the machine's input gradually, as execution of the current instruction progresses. Execution of instruction i progresses through states $\sigma \in \mathcal{S}$ of the control automaton from Fig. 29 in cycles

  $$T(\sigma,i) = \{t \mid I(t) = i \wedge fin(\sigma,t)\}.$$

  The latter cycles (if they exist) are provably unique (see Sect. 6.3.2, equation 19).

  $$\tau(\sigma,i) = \begin{cases} \varepsilon\, T(\sigma,i) & T(\sigma,i) \neq \emptyset \\ -1 & \text{otherwise} \end{cases}$$

  Note that if control state σ is not visited throughout execution of instruction i, cycle $\tau(\sigma,i)$ is simply chosen to be -1. For convenience we abbreviate

  $$\tau_{YT}(t) = \tau(YT, I(t))$$

  and specify the first two components of the oracle input for step $NS(t)$ as follows.

  $$s(NS(t)).w_Y = \begin{cases} mmu_Y^{\tau_{YT}(t)}.wout & \tau_{YT}(t) \geq 0 \wedge mmu_Y^{\tau_{YT}(t)}.treq \\ \perp & \text{otherwise} \end{cases}$$

  Thus, walk w_Y is taken from the outputs of mmu_Y once instruction i leaves the corresponding translation state. Finally, for the external event signals we specify

  $$s(NS(t)).eev = eev_*^t$$

  where the interrupts sampled at the end of the instruction execution (eev_*^t) are *not* the same interrupts that were sampled throughout the cycles of instruction execution ($eev(t)$):

  $$eev_*^t \;=\; eev(t) \wedge (idle(t) \circ 0).$$

According to Sect. 6.1.2, the last definition (for the processor core step) works for the external interrupts. For *reset* it works by assumption that the hardware reset is never active in cycles $t > 0$.

7.2 Developing Formalism

This technical section we spend to develop the machinery necessary to prove the simulation theorem from Sect. 7.1.3. In particular, below we introduce functions $I(t)$ and $i(t)$, which will be heavily used in the remainder of this chapter. Also, in Sect. 7.2.2 we establish an important relationship between (scheduling) function $I(t)$ and steps of the ISA computation.

7.2.1 Scheduling Function

In order to prove the correctness theorem for the sequential machine we require the following auxiliary machinery. We introduce a simple scheduling function for the sequential machine which keeps track of the number of instructions executed (processor core steps performed) up to hardware cycle t.

$$I(0) = 0$$
$$I(t+1) = \begin{cases} I(t)+1 & cstep(t) \\ I(t) & \text{otherwise} \end{cases}$$

We also require function *pseq*, which as usual maps the local processor instruction indices to the global step numbers. The definition is straightforward.

$$\begin{aligned} pseq(0) &= \min\{n \in \mathbb{N} \mid s(n) \in \Sigma_{core}\} \\ pseq(m+1) &= \min\{n \in \mathbb{N} \mid s(n) \in \Sigma_{core} \wedge n > pseq(m)\} \end{aligned}$$

The index of the configuration in which instruction $I(t)$ is executed we abbreviate by

$$i(t) = pseq(I(t)).$$

In the correctness proof below in this chapter we extensively argue about signals in ISA configuration n. For convenience we abbreviate these signals as

$$Z^n_\sigma = Z(c^n, s(n))$$

where $s(n)$ denotes the external input provided by the stepping function for step n. With these two shorthands we save considerable amount of space (and brackets). For instance, when we argue about signals in configuration in which instructions are executed, we write

$$Z^{i(t)}_\sigma = Z(c^{pseq(I(t))}, s(pseq(I(t)))).$$

7.2.2 Relating Global Steps with Scheduling Function

In this section we establish several important relations between the number of instructions executed up to hardware cycle t and the number of steps performed up to hardware cycle t. A technical lemma follows directly from the definitions above.

Lemma 64.

$$cstep(t) \;\rightarrow\; pseq(I(t)) = NS(t)$$

Proof of lemma 64. Consider two functions

$$\begin{aligned} f_1(t) &= pseq(I(t)) \\ f_2(t) &= NS(t) \end{aligned}$$

which are defined on domain

$$D = \{t \in \mathbb{N} \mid cstep(t)\}.$$

Since both functions

$$f_i :\; D \rightarrow R_i$$

are strictly increasing it suffices to show that their ranges are equal. We argue

$$\begin{aligned} R_1 &= \{pseq(I(t)) \mid t \in D\} \\ &= \{pseq(I(t)) \mid cstep(t)\} \\ &= \{pseq(i) \mid i \in \mathbb{N}\} \end{aligned}$$

and

$$\begin{aligned} R_2 &= \{NS(t) \mid t \in D\} \\ &= \{NS(t) \mid cstep(t)\} \\ &= \{n \in \mathbb{N} \mid s(n) \in \Sigma_{core}\}. \end{aligned}$$

Let us denote the n-th minimum element of set S by $\min\limits_n S$. Using sorted sequences

$$\vec{R}_i[n] = \min_n \; R_i$$

one easily obtains by induction on $n \in \mathbb{N}$:

$$\vec{R}_1[n] = \vec{R}_2[n].$$

From the monotonicity of function *pseq* we obviously have

$$\vec{R}_1[n] = pseq(n).$$

For the induction base ($n = 0$) we argue as follows.

$$\begin{aligned}\vec{R}_2[0] &= \min\ R_2\\ &= \min\{n \in \mathbb{N} \mid s(n) \in \Sigma_{core}\}\\ &= pseq(0)\end{aligned}$$

For the induction step from n to $n+1$ we show the following.

$$\begin{aligned}\vec{R}_2[n+1] &= \min_{n+1}\ R_2\\ &= \min\ (R_2 \setminus \bigcup_{k \le n} \vec{R}_2[k])\\ &= \min\{m \in \mathbb{N} \setminus \bigcup_{k \le n} \vec{R}_2[k] \mid s(m) \in \Sigma_{core}\}\\ &= \min\{m \in \mathbb{N} \mid s(m) \in \Sigma_{core} \wedge m > \vec{R}_1[n]\}\\ &= pseq(n+1) \qquad \square\end{aligned}$$

Using the latter lemma, for the walks passed to the specification machine one can easily derive the following result.

$$w_{Y\ \sigma}^{i(t)} = \begin{cases} mmu_Y^{\tau_{YT}(t)}.wout & \tau_{YT}(t) \ge 0 \wedge mmu_Y^{\tau_{YT}(t)}.treq \\ \bot & \text{otherwise} \end{cases} \tag{21}$$

Another technical lemma we need in the correctness proof below is the following.

Lemma 65.

$$pseq(I(t)) \ge NS(t)$$

Proof of lemma 65. Consider a pair of hardware cycles t_1, t_2 s.t.

$$cstep(t_1) \wedge cstep(t_2) \wedge (t \in (t_1, t_2) \rightarrow /cstep(t)).$$

Directly from the definition of the scheduling function we conclude:

$$\begin{aligned}\forall t \in (t_1, t_2) :\ I(t) &= I(t+1)\\ &= I(t_2).\end{aligned}$$

Using that function NS by definition is monotonically increasing, we argue:

$$\begin{aligned}\forall t \in (t_1, t_2) :\ pseq(I(t)) &= pseq(I(t_2))\\ &= NS(t_2) \qquad \text{(lemma 64)}\\ &\ge NS(t). \qquad \square\end{aligned}$$

In the last lemma of this section we argue that starting from (global) step $NS(t)$ up to (global) step $pseq(I(t))$, in which the current instruction is executed, no processor core steps are performed.

Lemma 66.

$$\forall t :\ n \in [NS(t) : pseq(I(t)) - 1]\ \rightarrow\ s(n) \notin \Sigma_{core}$$

Proof of lemma 66. We prove the statement by induction on n, where n denotes the length of sub-interval

$$[NS(t) : NS(t) + n - 1] \subseteq [NS(t) : pseq(I(t)) - 1].$$

The base case holds trivially since for $n = 0$ there is nothing to show. For the induction step from $n-1$ to

$$n \le pseq(I(t)) - NS(t)$$

we argue by contradiction as follows. Assume that $NS(t)+n$ is a step of the processor core:

$$s(NS(t)+n) \in \Sigma_{core}.$$

Obviously, there is a hardware cycle t' in which this step is performed, i.e.,

$$\exists t' : NS(t') = NS(t)+n.$$

From the monotonicity of function NS we immediately conclude

$$t' \ge t.$$

And from the monotonicity of function $pseq$ we derive a contradiction:

$$\begin{aligned} NS(t)+n &= NS(t') \\ &= pseq(I(t')) \qquad \text{(lemma 64)} \\ &\ge pseq(I(t)) \\ &> pseq(I(t)) - 1. \end{aligned}$$

□

7.3 Correctness Proof

At this point we finally have the machinery to show the correctness of the sequential machine with NAT. Utilizing all results obtained in this chapter, we turn the proof into a simple bookkeeping exercise; the proof sketch occupies less than two pages. The proof is obviously by induction on the number of hardware cycles t. Moreover, since in Chap. 5 we have already shown (lemma 36) that the translations in mmu_Y are contained in the TLB component of general computation $\tilde{c}_Y$, defined earlier in Sect. 3.4.

$$\exists \tilde{c}_Y^0 \; \forall t : sim_{tlb}(mmu_Y^t, \tilde{c}_Y^t) \tag{22}$$

Recall, stepping function $\tilde{s}_Y$ for general computation $\tilde{c}_Y$ was provided in Chap. 5. For convenience, we rewrite the latter definition such that it better matches the hardware descriptions of the sequential machine.

$$taddy_{Y\,X}(t) \;\rightarrow\; \tilde{s}_Y(t) = \begin{cases} (winit, upa_X(mmu_Y^t), mmu_Y^t.pto_X) & winit_X(mmu_Y^t) \\ (wext, mmu_Y^t.w_X, pte_X(mmu_Y^t)) & \text{otherwise} \end{cases}$$

$$tdrop_Y(t) \;\rightarrow\; \tilde{s}_Y(t) = \begin{cases} (drop, mmu_Y^t.inva) & mmu_Y^t.invlpg \\ (drop, mmu_Y^t.inva.vm) & mmu_Y^t.vmflush \\ (drop, all) & \text{otherwise} \end{cases}$$

Therefore, to incorporate the results of Chap. 5, it suffices to prove the following.

$$\exists c^0 \; \forall t : sim_{p+m}(h^t, c^{NS(t)}) \wedge sim_{tlb}^{\text{ISA}}(\tilde{c}_I^t, c^{NS(t)}) \wedge sim_{tlb}^{\text{ISA}}(\tilde{c}_E^t, c^{NS(t)})$$

Finally, we extend the induction hypothesis with an additional term for simulation of implementation registers, which we require to hold between configurations h^t and $c^{i(t)}$.

$$sim_{inv}(h^t, c^{i(t)})$$

The latter auxiliary relation is necessary to perform the proof.

Simulation of Implementation Registers

We can formulate the simulation relation for implementation registers as follows.
$sim_{inv}(h,c) \equiv$

1. $IF \le cs(h) \wedge used(w_I, c, x) \;\rightarrow\; h.w_I = x.w_I$
2. $ET \le cs(h) \wedge used(I, c, x) \;\rightarrow\; h.I = I(c,x)$
3. $EX \le cs(h) \wedge used(w_E, c, x) \;\rightarrow\; h.w_E = x.w_E$

Invariants

As additional proof goals we require the following invariants to hold.

Invariant 6.

$$IF \le cs(t) \wedge used(w_I)^{i(t)}_{\sigma} \ \rightarrow \ h^t.w_I \in \begin{cases} c^{i(t)}.tlb^{\circ} & user(t) \\ c^{i(t)}.tlb & guest(t) \end{cases}$$

$$EX \le cs(t) \wedge used(w_E)^{i(t)}_{\sigma} \ \rightarrow \ h^t.w_E \in \begin{cases} c^{i(t)}.tlb^{\circ} & user(t) \\ c^{i(t)}.tlb & guest(t) \end{cases}$$

Invariant 7.

$$IF \le cs(t) \wedge used(w_I)^{i(t)}_{\sigma} \ \rightarrow \ match(trq^{i(t)}_{I\ \sigma}, h^t.w_I)$$
$$EX \le cs(t) \wedge used(w_E)^{i(t)}_{\sigma} \ \rightarrow \ match(trq^{i(t)}_{E\ \sigma}, h^t.w_E)$$

Induction Base

For the induction base ($t = 0$) we argue as follows.

- simulation $sim^{\text{ISA}}_{tlb}(\tilde{c}^0_Y, c^{NS(0)})$ we obtain by constructing an appropriate ISA configuration c^0 from the general TLB configuration $\tilde{c}^0_Y$:
$$c^0.tlb = \emptyset \supseteq \emptyset = \tilde{c}^0_Y.tlb.$$
- simulation $sim_{p+m}(h^0, c^{NS(0)})$ we obtain as usual by constructing an appropriate ISA configuration c^0 from the hardware configuration h^0:
$$\begin{aligned} c^0.(pc, dpc, ddpc) &= h^0.(pc, dpc, ddpc) = (8_{32}, 4_{32}, 0_{32}) \\ c^0.(gpr, spr) &= h^0.(gpr, spr) \\ \ell(c^0.m) &= m(h^0). \end{aligned}$$
- simulation $sim_{inv}(h^0, c^{i(0)})$ holds trivially since the machine's control automaton starts in its idle state:
$$cs(0) = idle.$$

Induction Step

For the induction step from t to $t+1$ we show

$$sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_Y, c^{NS(t+1)}),\ sim_{p+m}(h^{t+1}, c^{NS(t+1)}) \text{ and } sim_{inv}(h^{t+1}, c^{i(t+1)})$$

by case split i) on the number of steps performed in cycle t ($ns(t)$) and ii) on the types of these steps (if there are any). Recall, by lemma 63 we know that

$$ns(t) \le 1.$$

- $ns(t) = 0$. No steps are performed in cycle t. Simulations
$$sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_Y, c^{NS(t+1)}) \text{ and } sim_{p+m}(h^{t+1}, c^{NS(t+1)})$$
of the TLBs and processor resp. hold trivially, since none of the visible data structures are updated in cycle t. Simulation
$$sim_{inv}(h^{t+1}, c^{i(t+1)})$$
of the implementation registers we cover in Sect. 7.3.3 below.

- $tstep(t) = 1$. One of the MMUs performs a TLB step in cycle t. We show simulation
$$sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_{Y}, c^{NS(t+1)})$$
of the TLBs in Sect. 7.4.1 below. Simulations
$$sim_{p+m}(h^{t+1}, c^{NS(t+1)}) \text{ and } sim_{inv}(h^{t+1}, c^{i(t+1)})$$
of the processor and implementation registers resp. hold trivially. Note, configuration of the processor changes only at the processor core steps. Also, using lemma 49, we argue that the implementation registers do not change either.
- $cstep(t) = 1$. Processor core performs a step in cycle t. For simulation
$$sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_{Y}, c^{NS(t+1)})$$
of the TLBs we split cases on whether an invalidating instruction is executed in cycle t ($istep(t)$). In Sect. 7.4.2 we present an argument for the case in which such an instruction is executed. Otherwise, the latter simulation holds trivially. Simulation
$$sim_{p+m}(h^{t+1}, c^{NS(t+1)})$$
of the processor boils down to a simple bookkeeping exercise; it is covered in Sect. 7.3.2. Finally, simulation
$$sim_{inv}(h^{t+1}, c^{i(t+1)})$$
for the implementation registers holds trivially, since according to lemma 47 the control goes to the idle state from cycle t and there is nothing to show in cycle $t+1$.

The invariants are shown in Sect. 7.3.4; the guard conditions are verified in Sect. 7.5. Following this sketch we finish proving correctness of the sequential machine in the forthcoming subsections.

7.3.1 Interrupt Processing

With the next lemmas we claim that all interrupts which were processed by hardware throughout execution of the current instruction were processed correctly.

Lemma 67.
$$\forall k \leq \max \mathcal{J}(cs(t)) : \ mca(t)[k] = mca^{i(t)}_{\sigma}[k]$$

Proof of lemma 67. For the external interrupt signal ($k = 1$), we split cases on whether a processor core step is performed in cycle t.

- $cstep(t) = 1$. In this case we trivially obtain:
$$\begin{aligned} e(t) &= idle(t) \wedge eev(t)[1] && \text{(definition)} \\ &= s(NS(t)).eev[1] && \text{(stepping)} \\ &= eev^{i(t)}_{\sigma}[1] && \text{(lemma 64)} \\ &= e^{i(t)}_{\sigma} && \text{(definition).} \end{aligned}$$
Moreover, for the interrupt mask signal we show:
$$\begin{aligned} imask(t)[1] &= h^{t}.sr[1] && \text{(definition)} \\ &= c^{NS(t)}.sr[1] && \text{(IH)} \\ &= c^{i(t)}.sr[1] && \text{(lemma 64)} \\ &= imask^{i(t)}_{\sigma}[1] && \text{(definition).} \end{aligned}$$
Combining the latter two arguments clearly gives the claim for the first bit.
$$mca(t)[1] = mca^{i(t)}_{\sigma}[1]$$

- $cstep(t) = 0$. This case turns out to be more complicated. First, by contradiction

$$\begin{aligned} mca(t)[1] &\rightarrow jisr(t) \wedge /cont(t) \\ &\rightarrow /exec(t) \\ &\rightarrow cstep(t) \end{aligned}$$

we derive $exec(t)$ and

$$0 = mca(t)[1] = h^t.sr[1] \wedge e(t).$$

Note, provided that the external interrupts are not masked in the SPR ($h^t.sr[1]$), the external interrupt signal ($e(t)$) must be low in cycle t if the machine resides in the *idle* state (since otherwise a core step is performed). Outside the *idle* state, the external interrupts are not visible by the hardware (Sect. 6.1.2), and therefore are not passed to the specification machine (Sect. 7.1.4). Thus, it suffices to show

$$e_\sigma^{i(t)} = 0.$$

For the first cycle $t' > t$ when a processor core step is performed

$$t' = \min\{\tilde{t} > t \mid cstep(\tilde{t})\}$$

we clearly have

$$/idle(t') \wedge cstep(t').$$

Analogous to the first case ($cstep(t)$), for the interrupt signal in cycle t' we derive the following.

$$e(t') \ = \ e_\sigma^{i(t')} \tag{23}$$

Since processor core steps are not performed in cycles $[t : t'-1]$, we conclude

$$\begin{aligned} e_\sigma^{i(t)} &= e_\sigma^{i(t')} && \text{(definition)} \\ &= e(t') && \text{(equation 23)} \\ &= 0 && \text{(definition)} \end{aligned}$$

and the claim follows.

Since all other interrupts are *not* maskable, we argue in the remainder of this proof only about the cause signals. Thus, for the misalignment on fetch ($k = 2$) we obtain:

$$\begin{aligned} malf(t) &\equiv h^t.ddpc[1:0] \neq 0_2 && \text{(definition)} \\ &\equiv c^{NS(t)}.ddpc[1:0] \neq 0_2 && \text{(IH)} \\ &\equiv c^{i(t)}.ddpc[1:0] \neq 0_2 && \text{(lemma 66)} \\ &\equiv malf_\sigma^{i(t)} && \text{(definition).} \end{aligned}$$

First, using the arguments above for walk w_I we derive the following.

$$IF \leq cs(t) \ \rightarrow \ (\ used(w_I)_\sigma^{i(t)} \leftrightarrow /host(t)\ ) \tag{24}$$

Proof of equation 24.

$$\begin{aligned} /host(t) &\leftrightarrow /host(t) \wedge (mca(t)[2:0] = 0_3) && \text{(invariant 5.1)} \\ &\leftrightarrow /host_\sigma^{i(t)} \wedge (mca_\sigma^{i(t)}[2:0] = 0_3) && \text{(IH; lemma 66)} \\ &\leftrightarrow /host_\sigma^{i(t)} \wedge il_\sigma^{i(t)} > 2 && \text{(definition)} \\ &\leftrightarrow used(w_I)_\sigma^{i(t)} && \text{(definition)} \end{aligned}$$

□

Having the result above, we argue about the page faults on fetch ($k = 3$)

$$\begin{aligned} pff(t) &= /host(t) \wedge f(h^t.w_I) \qquad \text{(definition)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge f(h^t.w_I) \qquad \text{(equation 24)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge match(trq^{i(t)}_{I\,\sigma}, h^t.w_I) \wedge f(h^t.w_I) \qquad \text{(invariant 7)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge pfault(trq^{i(t)}_{I\,\sigma}, h^t.w_I) \qquad \text{(definition)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge pfault(trq^{i(t)}_{I\,\sigma}, w^{i(t)}_{I\,\sigma}) \qquad \text{(IH)} \\ &= pff^{i(t)}_{\sigma} \qquad \text{(definition)} \end{aligned}$$

and the general-protection faults on fetch ($k = 4$) resp.

$$\begin{aligned} gff(t) &= /host(t) \wedge /f(h^t.w_I) \wedge (11 \not\leq h^t.w_I.r[\mathtt{xu}]) \qquad \text{(definition)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge /f(h^t.w_I) \wedge (11 \not\leq h^t.w_I.r[\mathtt{xu}]) \qquad \text{(equation 24)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge match(trq^{i(t)}_{I\,\sigma}, h^t.w_I) \wedge /f(h^t.w_I) \wedge (11 \not\leq h^t.w_I.r[\mathtt{xu}]) \\ &= used(w_I)^{i(t)}_{\sigma} \wedge gfault(trq^{i(t)}_{I\,\sigma}, h^t.w_I) \qquad \text{(definition)} \\ &= used(w_I)^{i(t)}_{\sigma} \wedge gfault(trq^{i(t)}_{I\,\sigma}, w^{i(t)}_{I\,\sigma}) \qquad \text{(IH)} \\ &= gff^{i(t)}_{\sigma} \qquad \text{(definition)} \end{aligned}$$

For the misalignment on memory operation ($k = 8$) we similarly obtain:

$$\begin{aligned} malm(t) &\equiv mmask(t)[2:1] \wedge (ea(t)[1:0] \neq 0_2) \qquad \text{(definition)} \\ &\equiv mmask^{i(t)}_{\sigma}[2:1] \wedge (ea^{i(t)}_{\sigma}[1:0] \neq 0_2) \qquad \text{(IH)} \\ &\equiv mop^{i(t)}_{\sigma} \wedge (d^{i(t)}_{\sigma} \nmid \langle ea^{i(t)}_{\sigma} \rangle) \qquad \text{(lemma 8)} \\ &\equiv malm^{i(t)}_{\sigma} \qquad \text{(definition).} \end{aligned}$$

Analogously to equation 24, using the arguments above for walk w_E we derive the following.

$$EX \leq cs(t) \;\rightarrow\; (\; used(w_E)^{i(t)}_{\sigma} \leftrightarrow /host(t) \wedge mop(t) \;) \tag{25}$$

Finally, we again similarly argue about the page faults on memory operation ($k = 9$)

$$\begin{aligned} pfm(t) &= /host(t) \wedge mop(t) \wedge f(h^t.w_E) \qquad \text{(definition)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge f(h^t.w_E) \qquad \text{(equation 25)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge match(trq^{i(t)}_{E\,\sigma}, h^t.w_E) \wedge f(h^t.w_E) \qquad \text{(invariant 7)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge pfault(trq^{i(t)}_{E\,\sigma}, h^t.w_E) \qquad \text{(definition)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge pfault(trq^{i(t)}_{E\,\sigma}, w^{i(t)}_{E\,\sigma}) \qquad \text{(IH)} \\ &= pfm^{i(t)}_{\sigma} \qquad \text{(definition)} \end{aligned}$$

and the general-protection-faults on memory operation ($k = 10$) resp.

$$\begin{aligned} gfm(t) &= /host(t) \wedge mop(t) \wedge /f(h^t.w_E) \wedge (1\mathtt{s} \not\leq h^t.w_E.r[\mathtt{uw}]) \qquad \text{(definition)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge /f(h^t.w_E) \wedge (1\mathtt{s} \not\leq h^t.w_E.r[\mathtt{uw}]) \qquad \text{(equation 25)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge match(trq^{i(t)}_{E\,\sigma}, h^t.w_E) \wedge /f(h^t.w_E) \wedge (1\mathtt{s} \not\leq h^t.w_E.r[\mathtt{uw}]) \\ &= used(w_E)^{i(t)}_{\sigma} \wedge gfault(trq^{i(t)}_{E\,\sigma}, h^t.w_E) \qquad \text{(definition)} \\ &= used(w_E)^{i(t)}_{\sigma} \wedge gfault(trq^{i(t)}_{E\,\sigma}, w^{i(t)}_{E\,\sigma}) \qquad \text{(IH)} \\ &= gfm^{i(t)}_{\sigma} \qquad \text{(definition)} \end{aligned}$$

where for the write-bit checked by the hardware we obviously have

$$\mathtt{s} \;\equiv\; s(t) \vee cas(t) \;=\; s^{i(t)}_{\sigma} \vee cas^{i(t)}_{\sigma}. \qquad \square$$

Processing of the remaining interrupt signals does not change compared to [LOP], and therefore is omitted. Assuming correctness of the simple circuits implementing the machine control logic, in particular the signals above, we argue further as follows.

Lemma 68. *On steps of the processor core ($cstep(t)$) the following holds:*

$$\begin{aligned} jisr(t) &= jisr_\sigma^{i(t)} \\ f1(mca(t)) &= f1(mca_\sigma^{i(t)}) \end{aligned}$$

Proof of lemma 68. First we consider the JISR signal.

$$\begin{aligned} jisr(t) &= \textstyle\bigvee_{k \le \max \mathcal{J}(cs(t))} mca(t)[k] && \text{(definition)} \\ &= \textstyle\bigvee_{k \le \max \mathcal{J}(cs(t))} mca_\sigma^{i(t)}[k] && \text{(lemma 67)} \end{aligned}$$

For the direction from left to right we clearly have:

$$\begin{aligned} jisr(t) &\rightarrow mca_\sigma^{i(t)}[10:0] \neq 0_{11} \\ &\rightarrow jisr_\sigma^{i(t)} \qquad \text{(definition).} \end{aligned}$$

For the opposite direction we first recall that

$$cstep(t) \wedge /jisr(t) \rightarrow cs(t) = EX.$$

Using that we trivially conclude (by contradiction):

$$\begin{aligned} /jisr(t) &\rightarrow mca_\sigma^{i(t)}[10:0] = 0_{11} \\ &\rightarrow /jisr_\sigma^{i(t)} \qquad \text{(definition).} \end{aligned}$$

For the masked cause, from lemma 67 we have

$$mca(t)[k^*:0] = mca_\sigma^{i(t)}[k^*:0]$$

for

$$k^* = \max \mathcal{J}(cs(t)).$$

Next we split cases on the value of the JISR signal.

- $jisr(t) = 1$. In case of an active JISR we know that

  $$\exists k \le k^* : \; mca(t)[k] \wedge mca_\sigma^{i(t)}[k].$$

  Using properties of the *first one* circuit we easily derive

  $$\begin{aligned} f1(mca(t)) &= 0_{10-k^*} \circ f1(mca(t)[k^*:0]) \\ &= 0_{10-k^*} \circ f1(mca_\sigma^{i(t)}[k^*:0]) \\ &= f1(mca_\sigma^{i(t)}). \end{aligned}$$

- $jisr(t) = 0$. In case of an inactive JISR we simply have

  $$k^* = 10 \wedge \forall k \le k^* : \; /mca(t)[k] \wedge /mca_\sigma^{i(t)}[k]$$

  and the statement follows.

  $$\begin{aligned} f1(mca(t)) &= 0_{32} \\ &= f1(mca_\sigma^{i(t)}) \end{aligned}$$

□

7.3.2 Instruction Execution

In Sect. 7.3.1 we deduced that the interrupts registered by hardware are identical to the interrupts registered in the specification on execution of the current instruction. In this section we argue that our sequential implementation executes instructions correctly. Instructions are executed in cycles t in which the processor core steps are performed ($cstep(t)$). In case no interrupts occurred in cycle t by lemma 68 we know

$$jisr(t) = 0 = jisr_{\sigma}^{i(t)}$$

and the current instruction is executed in a regular fashion. Correctness for this case was established for sequential machines a number of times; similar proofs can be found in [PBLS16], [Sch14a], and [Sch16b].

Interrupted Execution

Next we consider the case in which a non-continue type interrupt was discovered in cycle t, i.e., in which we have

$$jisr(t) \wedge /cont(t).$$

On non-continue type interrupts execution of the current instruction is terminated and no visible data structures except for programs counters and special purpose registers are updated. For the program counters we trivially derive:

$$\begin{aligned} h^{t+1}.pc &= 8_{32} = c^{NS(t+1)}.pc \\ h^{t+1}.dpc &= 4_{32} = c^{NS(t+1)}.dpc \\ h^{t+1}.ddpc &= 0_{32} = c^{NS(t+1)}.ddpc. \end{aligned}$$

For the special purpose registers the arguments are close to trivial as well. For the status register (sr) we simply have

$$h^{t+1}.sr = 0_{32} = c^{NS(t+1)}.sr.$$

For the mode registers we apply lemma 64 and split cases on the value of signal

$$user(t) \wedge /icpt(t) \;\leftrightarrow\; user_{\sigma}^{NS(t)} \wedge /icpt_{\sigma}^{NS(t)}.$$

In case the latter signal is active, we argue about the nested mode register ($nmode$)

$$h^{t+1}.nmode = h^{t}.nmode[31:1] \circ 0 = c^{NS(t)}.nmode[31:1] \circ 0 = c^{NS(t+1)}.nmode$$

whereas for the ordinary mode register ($mode$) the argument it trivial.

$$h^{t+1}.mode = h^{t}.mode = c^{NS(t)}.mode = c^{NS(t+1)}.mode$$

Otherwise, in case an interrupt is triggered at the lower privilege levels (of guest or host) or in case of an intercept, the argument for the nested mode register repeats the corresponding argument for the ordinary mode register above, and vice versa.

$$\begin{aligned} h^{t+1}.nmode = h^{t}.nmode &= c^{NS(t)}.nmode = c^{NS(t+1)}.nmode \\ h^{t+1}.mode = h^{t}.mode[31:1] \circ 0 &= c^{NS(t)}.mode[31:1] \circ 0 = c^{NS(t+1)}.mode \end{aligned}$$

For most of the “exception” registers

$$Z \in \{sr, pc, dpc, ddpc, mode, nmode\}$$

we simply have

$$h^{t+1}.eZ = h^{t}.Z = c^{NS(t)}.Z = c^{NS(t+1)}.eZ.$$

Update of the exception cause register (eca) is justified by application of lemmas 68 and 64:

$$h^{t+1}.eca = 0_{21} \circ f1(mca(t)) = 0_{21} \circ f1(mca_{\sigma}^{i(t)}) = c^{NS(t+1)}.eca.$$

Finally, update of the exception data register (*edata*) depends on the control state ($cs(t)$) in which the processor core step is performed. On JISR by applying lemmas 68 and 64 we easily derive:

$$IF < cs(t) \leftrightarrow mca(t)[4:0] = 0_5 \leftrightarrow mca_{\sigma}^{i(t)}[4:0] = 0_5 \leftrightarrow 4 < il_{\sigma}^{i(t)} \leftrightarrow fetch_{\sigma}^{NS(t)}.$$

If the latter signal is active, we have

$$h^{t+1}.edata = ea(t) = ea_{\sigma}^{NS(t)} = c^{NS(t+1)}.edata.$$

Otherwise, we simply have

$$h^{t+1}.edata = 0_{32} = c^{NS(t+1)}.edata.$$

The remaining special purpose registers do not change.

Uninterrupted Execution

In case a highest priority interrupt which occurred throughout the cycles of execution of the current instruction is of type continue, in cycle t we clearly have

$$jisr(t) \wedge cont(t)$$

and execution of the current instruction is not interrupted ($exec(t)$). First of all, from the hardware construction we know that the processor core step is performed in the execution state.

$$cs(t) = EX$$

In contrast to the previous case, with continue type interrupts the results of instruction execution must be written to their target data structures. Since in our simple design the only two sources for the continue type interrupts are resp. the system calls and the arithmetics overflows, we proceed as follows.

In case of a system call (*sysc*), no visible data structures are updated explicitly (by the system call instruction).[1] In case of the arithmetic overflow (*ovf*), which occurs on execution of arithmetic instructions, the intermediate result C is written into the target general purpose register. Using lemma 64 we argue for the C-address

$$xad = xad(t) = xad_{\sigma}^{NS(t)}$$

and for the target GPR register.

$$h^{t+1}.gpr(xad) = C(t) = C_{\sigma}^{NS(t)} = c^{NS(t+1)}.gpr(xad)$$

The program counters are updated with the initial values exactly like in the previous case (for the non-continue type interrupts). Most of the special purpose registers are updated exactly like in the previous case as well. The only difference is the exception program counters. For the latter registers we argue using the induction hypothesis simply as follows.

$$\begin{aligned} h^{t+1}.epc &= nextpc(t) = nextpc_{\sigma}^{NS(t)} = c^{NS(t+1)}.epc \\ h^{t+1}.edpc &= h^{t}.pc = c^{NS(t)}.pc = c^{NS(t+1)}.edpc \\ h^{t+1}.eddpc &= h^{t}.dpc = c^{NS(t)}.dpc = c^{NS(t+1)}.eddpc \end{aligned}$$

Note, in the argument for the exception PC above we additionally require lemma 64 to justify correctness of the *nextpc* computation.

[1] I.e., the general purpose register file and the memory are not affected; PCs are set to point to three consecutive words after *sisr* (*start of the interrupt service routine*), and the special purpose registers are updated according to the specification of continue type interrupts.

7.3.3 Implementation Registers

In order to show

$$sim_{inv}(h^{t+1}, c^{i(t+1)})$$

we split cases on the next control state ($cs(t+1)$). Obviously we assume

$$cs(t+1) \neq cs(t)$$

since otherwise nothing changes and the simulation holds trivially.

- $cs(t+1) \in \{idle, IT\}$. The simulation relation holds trivially for this case.
- $cs(t+1) = IF$. In a dedicated section below we show

$$used(w_I)_\sigma^{i(t+1)} \rightarrow h^{t+1}.w_I = w_{I\ \sigma}^{i(t+1)}.$$

- $cs(t+1) = ET$. In a dedicated section below we show

$$used(I)_\sigma^{i(t+1)} \rightarrow h^{t+1}.I = I_\sigma^{i(t+1)}.$$

- $cs(t+1) = EX$. In a dedicated section below we show

$$used(w_E)_\sigma^{i(t+1)} \rightarrow h^{t+1}.w_E = w_{E\ \sigma}^{i(t+1)}.$$

For cases in which

$$cs(t+1) \in \{IF, ET, EX\}$$

we argue that

$$i(t+1) = i(t)$$

since no processor core steps are performed. Therefore, the portions of the simulation relation which we did not mention above hold trivially by induction.

Instruction Address Translation

Below we show that register w_I if used for translation of the instruction address, contains a proper translation in cycle $t+1$. Recall, we show correctness for register w_I only in those cycles in which

$$IT(t) \wedge /idle(t+1).$$

Therefore, using lemma 47 we conclude there is no processor core step in cycle t, which implies

$$i(t+1) = i(t). \tag{26}$$

From the used predicate we derive:

$$\begin{aligned} used(w_I)_\sigma^{i(t+1)} &\leftrightarrow used(w_I)_\sigma^{i(t)} && \text{(equation 26)} \\ &\rightarrow /host_\sigma^{i(t)} && \text{(definition)} \\ &\leftrightarrow /host(c^{NS(t)}) && \text{(lemma 66)} \\ &\leftrightarrow /host(t) && \text{(IH).} \end{aligned}$$

Moreover, we have $exec(t)$, and thus $mmu_I^t.treq$, which allows us to show the claim.

$$\begin{aligned} h^{t+1}.w_I &= mmu_I^t.wout && \text{(interconnect)} \\ &= w_{I\ \sigma}^{i(t)} && \text{(equation 21)} \\ &= w_{I\ \sigma}^{i(t+1)} && \text{(equation 26)} \end{aligned}$$

Instruction Fetch

We use this section to show that the instruction fetched in hardware cycle t is the instruction executed in ISA configuration $i(t)$. Recall, pma_I is the physical memory address of the fetched instruction. In case the address translation is used we derive:

$$\begin{aligned} pma_I(t) &= h^t.w_I.ba \circ ia(t).po \qquad \text{(interconnect)} \\ &= h^t.w_I.ba \circ ia(c^{NS(t)}).po \qquad \text{(IH)} \\ &= h^t.w_I.ba \circ ia_\sigma^{i(t)}.po \qquad \text{(lemma 66)} \\ &= tma(asid_\sigma^{i(t)} \circ ia_\sigma^{i(t)}, w_{I\ \sigma}^{i(t)}) \qquad \text{(equation 32; invariant 7; IH)} \\ &= pma_{I\ \sigma}^{i(t)} \qquad \text{(definition).} \end{aligned}$$

Otherwise, without the address translation, we have:

$$\begin{aligned} pma_I(t) &= ia(t) \qquad \text{(interconnect)} \\ &= ia(c^{NS(t)}) \qquad \text{(IH)} \\ &= ia_\sigma^{i(t)} \qquad \text{(lemma 66)} \\ &= pma_{I\ \sigma}^{i(t)} \qquad \text{(definition).} \end{aligned}$$

Having that the address used by the hardware to fetch the instruction is the address used in the ISA, for the output of the instruction cache we show the following.

$$finex(IF,t) \ \rightarrow \ pdout(1)^t = imout_\sigma^{i(t)} \tag{27}$$

Proof of equation 27. According to lemmas from Sect. 6.2.4, we know there is an access to the instruction cache ending in cycle t (lemma 55)

$$\exists k : \ e(1,k) = t \wedge acc(1,k).r$$

and it is the only non-flushing access ending in cycle t (lemmas 54 and 55).

$$\forall j \in \{0,2,3\} \ \forall k : \ e(j,k) = t \rightarrow acc(j,k).f \tag{28}$$

For convenience we abbreviate

$$a = pma_I.l$$

and derive:

$$\begin{aligned} pdout(1)^t &= \Delta_M^{seq(1,k)}(m(h^0), acc')(a) \qquad \text{(lemma 52)} \\ &= \Delta_M^{NE(t)+n_I}(m(h^0), acc')(a) \qquad \text{(where } n_I \leq ne(t)) \\ &= \Delta_M^{NE(t)}(m(h^0), acc')(a) \qquad \text{(equation 28)} \\ &= m(h^t)(a) \qquad \text{(lemma 51)} \\ &= \ell(c^{NS(t)}.m)(a) \qquad \text{(IH)} \\ &= \ell(c^{i(t)}.m)(a) \qquad \text{(lemma 66)} \\ &= dataout(\ell(c^{i(t)}.m), iacc_\sigma^{i(t)}) \qquad \text{(definition)} \\ &= imout_\sigma^{i(t)} \qquad \text{(definition).} \end{aligned}$$

□

Hardware fetches instructions only in the absence of interrupts which prevent the fetch. Recall, we argue about the instruction register only in those cycles in which

$$IF(t) \wedge /idle(t+1).$$

Thus, if no interrupts occurred

$$\begin{aligned}
\mathit{finex}(IF,t) \;\rightarrow\; /jisr(t) &\leftrightarrow mca(t)[4:0]=0_5 && \text{(definition)}\\
&\leftrightarrow mca_{\sigma}^{i(t)}[4:0]=0_5 && \text{(lemma 67)}\\
&\leftrightarrow used(I)_{\sigma}^{i(t)} && \text{(definition)}\\
&\leftrightarrow used(I)_{\sigma}^{i(t+1)} && \text{(lemma 47.1),}
\end{aligned}$$

for the instruction register we have:

$$\begin{aligned}
h^{t+1}.I &= \begin{cases} pdout(1)^t_H & pma_I[2] \\ pdout(1)^t_L & \text{otherwise} \end{cases} && \text{(interconnect)}\\
&= \begin{cases} imout^{i(t)}_{\sigma\, H} & pma_I[2] \\ imout^{i(t)}_{\sigma\, L} & \text{otherwise} \end{cases} && \text{(equation 27)}\\
&= I^{i(t)}_{\sigma} && \text{(lemma 7)}\\
&= I^{i(t+1)}_{\sigma} && \text{(lemma 47.1).}
\end{aligned}$$

Effective Address Translation

The arguments here are completely analogous to those presented in the counterpart section on translation of the instruction address, and therefore are omitted.

7.3.4 Maintaining Invariants

In this section we split cases on the next control state ($cs(t+1)$). Obviously we assume

$$cs(t+1) \neq cs(t)$$

since otherwise nothing changes and the invariants hold trivially.

- $cs(t+1) \in \{idle, IT\}$. The simulation relation holds trivially for this case.
- $cs(t+1) = IF$. In a dedicated section below we show

  $$match(trq^{i(t+1)}_{I\;\sigma}, h^{t+1}.w_I) \quad \text{and} \quad h^{t+1}.w_I \in \begin{cases} c^{i(t+1)}.tlb & guest(t+1) \\ c^{i(t+1)}.tlb^{\circ} & user(t+1) \end{cases}$$

  in case translation w_I for the instruction address is used in step $i(t+1)$.
- $cs(t+1) = ET$. The simulation relation holds trivially for this case.
- $cs(t+1) = EX$. In a dedicated section below we show

  $$match(trq^{i(t+1)}_{E\;\sigma}, h^{t+1}.w_E) \quad \text{and} \quad h^{t+1}.w_E \in \begin{cases} c^{i(t+1)}.tlb & guest(t+1) \\ c^{i(t+1)}.tlb^{\circ} & user(t+1) \end{cases}$$

  in case translation w_E for the effective address is used in step $i(t+1)$.

For cases in which

$$cs(t+1) \in \{EX\}$$

we argue that

$$i(t+1) = i(t)$$

since no processor core steps are performed. Therefore, the portions of the invariants which we did not mention above hold trivially by induction.

Translation for Instruction Address

That register w_I is updated with a walk that is contained in the specification TLB (when the processor core makes a step) we argue as follows:

$$\begin{aligned}
h^{t+1}.w_I &= mmu_I^t.wout \qquad \text{(interconnect)}\\
&\in \begin{cases} tlb_{I\,G}(t) & mmu_I^t.upa \in A_G \\ tlb_{I\,U}(t) & mmu_I^t.upa \in A_U \end{cases} \qquad \text{(interconnect; spec.)}\\
&\subseteq \begin{cases} c^{NS(t)}.tlb & guest(t) \\ c^{NS(t)}.tlb^\circ & user(t) \end{cases} \qquad \text{(equation 22; IH)}\\
&\subseteq \begin{cases} c^{i(t)}.tlb & guest(t) \\ c^{i(t)}.tlb^\circ & user(t) \end{cases} \qquad \text{(lemma 66)}\\
&= \begin{cases} c^{i(t+1)}.tlb & guest(t+1) \\ c^{i(t+1)}.tlb & user(t+1) \end{cases} \qquad \text{(lemma 47.1).}
\end{aligned}$$

Note that we show correctness for implementation register $h.w_I$ only in those cycles in which

$$IT(t) \wedge /idle(t+1).$$

Therefore, the execution mode does not change in cycle t

$$mode(t+1) = mode(t)$$

which justifies the last line in the proof above.
That this walk matches the address of the executed instruction is established below.

$$\begin{aligned}
h^{t+1}.w_I.upa &= mmu_I^t.wout.upa \qquad \text{(interconnect)}\\
&= mmu_I^t.upa \qquad \text{(specification)}\\
&= asid(t) \circ ia(t).pa \qquad \text{(interconnect)}\\
&= asid(c^{NS(t)}) \circ ia(c^{NS(t)}).pa \qquad \text{(IH)}\\
&= asid_\sigma^{i(t)} \circ ia_\sigma^{i(t)}.pa \qquad \text{(lemma 66)}\\
&= asid_\sigma^{i(t+1)} \circ ia_\sigma^{i(t+1)}.pa \qquad \text{(lemma 47.1).}
\end{aligned}$$

That this walk is faulty or complete is one of the properties of the nested MMU.

Translation for Effective Address

The arguments here are completely analogous to those presented above on translation for the instruction address, and therefore are omitted.

7.4 Correctness for TLBs

In this section we prove several results which establish correctness for the TLBs while the MMUs are serving translation and invalidation queries. We prove these results under the assumption that the MMU components are properly connected to i) the memory system on one hand and ii) the processor core on the other hand.

7.4.1 Address Translation

From the interface between the MMU and the memory in the following lemma we show that — after the memory access was performed — the hardware operates on the same page table entry as the specification machine does.

Lemma 69. *Assume that mmu_Y performs a memory access in cycle t ($wext(mmu_Y^t)$).*

$$pte(mmu_Y^t) = pte_\sigma^{NS(t)}$$

Proof of lemma 69. From the construction we know that the hardware performs extension of the *user walk* iff the nested control automaton of mmu_Y resides in state *fetch-pte*$_U$.

$$wext(mmu_Y^t) \ \rightarrow \ (\ mmu_Y^t.fetch\text{-}pte_U \leftrightarrow wext_U(mmu_Y^t)\ ) \tag{29}$$

First we argue about the page table entry addresses.

$$\begin{aligned}
ptea(mmu_Y^t) &= \begin{cases} ptea_U(mmu_Y^t) & wext_U(mmu_Y^t) \\ ptea_G(mmu_Y^t) & \text{otherwise} \end{cases} && \text{(definition; equation 29)} \\
&= \begin{cases} ptea(mmu_Y^t.w_U \circ mmu_Y^t.w_G) & wext_U(mmu_Y^t) \\ ptea(mmu_Y^t.w_G) & \text{otherwise} \end{cases} && \text{(lemma 20)} \\
&= ptea(s(NS(t))) \qquad \text{(stepping)}
\end{aligned}$$

For the output of cache ca_{YT} we derive the following.

$$wext(mmu_Y^t) \ \rightarrow \ pdout(2\mathbb{1}_{\{Y=E\}})^t = tmout_\sigma^{NS(t)} \tag{30}$$

Proof of equation 30. According to lemmas from Sect. 6.2.4 we know there is an access to cache ca_{YT} ending in cycle t (lemma 53)

$$\exists k:\ e(2\mathbb{1}_{\{Y=E\}},k) = t \wedge acc(2\mathbb{1}_{\{Y=E\}},k).r$$

and it is the only non-flushing access ending in cycle t (lemmas 54 and 55).

$$\forall j \neq 2\mathbb{1}_{\{Y=E\}}\ \forall k:\ e(j,k) = t \rightarrow acc(j,k).f \tag{31}$$

For convenience we abbreviate

$$a = mmu_Y^t.ma.l = ptea(mmu_Y^t).l = ptea(s(NS(t))).l$$

and derive:

$$\begin{aligned}
pdout(2\mathbb{1}_{\{Y=E\}})^t &= \Delta_M^{seq(2\mathbb{1}_{\{Y=E\}},k)}(m(h^0),acc')(a) && \text{(lemma 52)} \\
&= \Delta_M^{NE(t)+n_Y}(m(h^0),acc')(a) && \text{(where } n_Y \le ne(t)\text{)} \\
&= \Delta_M^{NE(t)}(m(h^0),acc')(a) && \text{(equation 31)} \\
&= m(h^t)(a) && \text{(lemma 51)} \\
&= \ell(c^{NS(t)}.m)(a) && \text{(IH)} \\
&= dataout(\ell(c^{NS(t)}.m), tacc_\sigma^{NS(t)}) && \text{(definition)} \\
&= tmout_\sigma^{NS(t)} && \text{(definition).}
\end{aligned}$$

□

Finally, for the page table entries we argue as follows.

$$\begin{aligned}
pte(mmu_Y^t) &= \begin{cases} pdout(2\mathbb{1}_{\{Y=E\}})_H^t & ptea(mmu_Y^t)[2] \\ pdout(2\mathbb{1}_{\{Y=E\}})_L^t & \text{otherwise} \end{cases} && \text{(interconnect)} \\
&= \begin{cases} tmout_{\sigma\,H}^{NS(t)} & ptea(s(NS(t)))[2] \\ tmout_{\sigma\,L}^{NS(t)} & \text{otherwise} \end{cases} && \text{(equation 30)} \\
&= pte_\sigma^{NS(t)} && \text{(lemma 14)}
\end{aligned}$$

□

In addition, we require that the data used to serve the translation queries are valid.

Lemma 70.

$$(pto, npto, asid)(t) = (pto, npto, asid)(c^{NS(t)})$$

The proof of lemma 70 is trivial, and therefore we omit it.

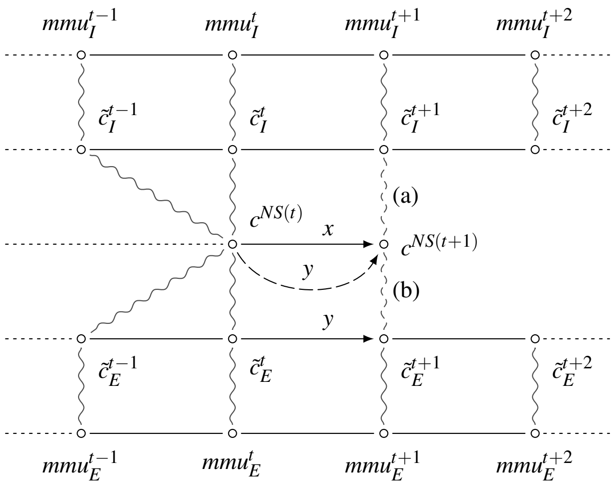


Fig. 31: TLB step performed by mmu_E in cycle t

Translation Queries

To show correctness of the translation queries we assume an active TLB transition, i.e., a TLB step, is performed by mmu_Y in cycle t ($tstep_Y(t)$). In the induction step we have to show both,

$$sim_{tlb}^{\text{ISA}}(\tilde{c}_I^{t+1}, c^{NS(t+1)}) \quad \text{and} \quad sim_{tlb}^{\text{ISA}}(\tilde{c}_E^{t+1}, c^{NS(t+1)}).$$

The proof is illustrated in Fig. 31, where the latter two goals are denoted resp. by (a) and (b). W.l.o.g. we assume that the step is performed by mmu_E, and proceed to show (b), since (a) turns to be completely trivial in this case.

As usual, we denote the input for the step performed in the general semantics by y, whereas the input of the corresponding step of the original semantics by x. Note, for the definitions of the stepping inputs below we refer to pages 151 and 147 resp.

$$\begin{aligned} y &= \tilde{s}_E(t) \\ x &= s(NS(t)) \end{aligned}$$

First we argue that the TLB step performed in the original semantics using input x is TLB equivalent to a step performed in the general semantics using input y. Aiming at application of lemma 17, according to Sect. 3.4.3, we proceed to show that the following conditions are satisfied by input y. We split cases on the step type:

- step of walk initialization.

$$winit_G(mmu_E^t) \rightarrow \begin{cases} y.upa &= vmid(c^{NS(t)}) \circ 0_8 \circ x.a \\ y.pto &= pto(c^{NS(t)}) \end{cases}$$

$$winit_U(mmu_E^t) \rightarrow \begin{cases} y.upa &= asid(c^{NS(t)}) \circ x.a \\ y.pto &= npto(c^{NS(t)}) \end{cases}$$

- step of walk extension.

$$wext_X(mmu_E^t) \rightarrow \begin{cases} y.pte &= pte_\sigma^{NS(t)} \\ y.w &= x.w_1 \end{cases}$$

All these conditions are established simply by unfolding the definitions and using the lemmas shown previously. For instance, on initialization of a user walk we argue as follows. For the page table origin used in the general semantics we have

$$\begin{aligned} y.pto &= mmu^t_E.pto_U && \text{(definition)} \\ &= npto(t) && \text{(interconnect)} \\ &= npto(c^{NS(t)}) && \text{(lemma 70)} \end{aligned}$$

whereas for the universal page address passed by input y we show

$$\begin{aligned} y.upa &= upa_U(mmu^t_E) && \text{(definition)} \\ &= mmu^t_E.upa.as \circ upa_U(mmu^t_E).pa && \text{(construction)} \\ &= asid(t) \circ upa_U(mmu^t_E).pa && \text{(interconnect)} \\ &= asid(c^{NS(t)}) \circ upa_U(mmu^t_E).pa && \text{(lemma 70)} \\ &= asid(c^{NS(t)}) \circ x.a && \text{(definition).} \end{aligned}$$

On steps of the walk extension we argue similarly. For the page table origin used in the general semantics we have

$$\begin{aligned} y.pte &= pte(mmu^t_E) && \text{(definition)} \\ &= pte^{NS(t)}_\sigma && \text{(lemma 69)} \end{aligned}$$

whereas for the universal walk passed by input y we show

$$\begin{aligned} y.w &= \begin{cases} mmu^t_E.w_G & wext_G(mmu^t_E) \\ mmu^t_E.w_U & \text{otherwise} \end{cases} && \text{(definition)} \\ &= x.w_1 && \text{(definition).} \end{aligned}$$

Even though the proofs turn to be completely straightforward, one has to repeat them for other machine types in order to rely on correctness of the MMU construction.
Now, using auxiliary configuration c' s.t.

$$c'.Z = \begin{cases} \delta_{tlb}(c^{NS(t)}.tlb, y) & Z = tlb \\ c^{NS(t)}.Z & \text{otherwise,} \end{cases}$$

we finish the proof of (b) as follows.

$$\begin{aligned} sim^{\text{ISA}}_{tlb}(\tilde{c}^t_E, c^{NS(t)}) &\rightarrow sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_E, c') && \text{(lemma 18)} \\ &\rightarrow sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_E, c^{NS(t+1)}) && \text{(lemma 17)} \end{aligned}$$

Finally, it remains to show (a). We obtain it directly from the induction hypothesis

$$sim^{\text{ISA}}_{tlb}(\tilde{c}^t_I, c^{NS(t)}) \rightarrow sim^{\text{ISA}}_{tlb}(\tilde{c}^{t+1}_I, c^{NS(t+1)})$$

since on steps of mmu_E we obviously have

$$\begin{aligned} \tilde{c}^{t+1}_I.tlb &= \tilde{c}^t_I.tlb \\ c^{NS(t+1)}.tlb &\supseteq c^{NS(t)}.tlb. \end{aligned}$$

7.4.2 Invalidation of TLB

To prove correctness of the invalidation queries we exploit that the following conditions are satisfied by the interface between the processor core and the MMU.

Lemma 71. *Assume that an invalidating instruction is executed in cycle t ($istep(t)$).*

$$mmu^t_Y.invlpg \leftrightarrow invlpg^{NS(t)}_\sigma \wedge \overline{user(c^{NS(t)})} \tag{1}$$

$$mmu^t_Y.vmflush \leftrightarrow flusht^{NS(t)}_\sigma \wedge guest(c^{NS(t)}) \tag{2}$$

$$mmu^t_Y.flush \leftrightarrow flusht^{NS(t)}_\sigma \wedge host(c^{NS(t)}) \tag{3}$$

Moreover, for the invalidation input of the TLB component we have:

$$mmu^t_Y.inva = inva^{NS(t)}_\sigma \tag{4}$$

The proof of lemma 71 is given below. For better presentation we merge parts 1–3.

Proof of lemmas 71.1–3. From simulation of implementation registers we immediately obtain that the hardware and the ISA execute the same instruction

$$h^t.I = I_\sigma^{i(t)}$$

since the invalidating steps of the processor core are possible only in state *EX*. From interconnect we know that the invalidating instruction executed by the processor core is executed uninterrupted.

$$\begin{aligned} istep(t) &\rightarrow cstep(t) \wedge exec(t) \qquad \text{(definition)} \\ &\rightarrow cstep(t) \wedge (cont(t) \vee /jisr(t)) \qquad \text{(definition)} \\ &\rightarrow exec_\sigma^{i(t)} \qquad \text{(lemma 68)} \end{aligned}$$

Having this, using correctness of the instruction decoder implementation we obtain the desired result just as we regularly do for other predicates and function fields [PBLS16]. Thus, for the invalidating instructions we conclude

$$\begin{aligned} invlpg(t) &\leftrightarrow invlpg_\sigma^{i(t)} \\ flusht(t) &\leftrightarrow flusht_\sigma^{i(t)} \end{aligned}$$

and the claim follows by lemma 64. □

Proof of lemma 71.4. The invalidation address is taken directly from the GPR. For the ASID to be invalidated we argue:

$$\begin{aligned} mmu_Y^t.inva.as &= \begin{cases} vmid(t) \circ A(t)[27:20] & guest(t) \\ A(t)[31:20] & \text{otherwise} \end{cases} \qquad \text{(interconnect)} \\ &= \begin{cases} vmid(c^{NS(t)}) \circ A_\sigma^{NS(t)}[27:20] & guest(c^{NS(t)}) \\ A_\sigma^{NS(t)}[31:20] & \text{otherwise} \end{cases} \qquad \text{(IH)} \\ &= inva_\sigma^{NS(t)}.as \qquad \text{(definition).} \end{aligned}$$

Similarly, for the invalidated page address we derive:

$$\begin{aligned} mmu_Y^t.inva.pa &= B(t)[31:12] \qquad \text{(interconnect)} \\ &= B_\sigma^{NS(t)}[31:12] \qquad \text{(IH)} \\ &= inva_\sigma^{NS(t)}.pa \qquad \text{(definition).} \end{aligned}$$
□

Invalidation Queries

To show correctness of the invalidation queries we assume a passive TLB transition, i.e., a processor core step executing an invalidating instruction, is performed in cycle t ($istep(t)$). In the induction step we have to show

$$sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^{t+1}, c^{NS(t+1)}).$$

The proof is illustrated in Fig. 32. In contrast to the translation queries here we do not have to differentiate between the MMUs, and can perform the proof for mmu_Y.

Just as before, we denote the input for the steps performed in the general semantics by y, whereas the input of the corresponding step of the original semantics by x. Again, for the definitions of the stepping inputs below we refer to pages 151 and 147 resp.

$$\begin{aligned} y &= \tilde{s}_Y(t) \\ x &= s(NS(t)) \end{aligned}$$

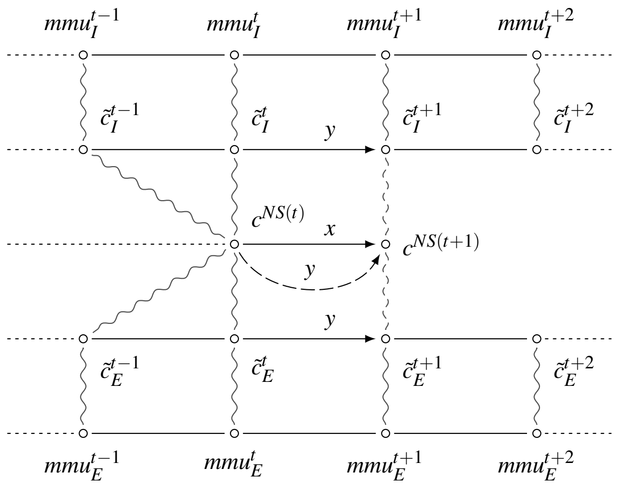


Fig. 32: Invalidating step (of the processor core) performed in cycle t

We argue that in configuration $NS(t)$ the step performed in the original semantics (using input x) is TLB equivalent to a step performed in the general semantics (using input y) in cycle t. Again aiming at application of lemma 17, according to Sect. 3.4.3, it suffices to show that the following conditions are satisfied by input y. As usual we split cases on the step type:

- invalidation of one translation.

$$/user(t) \wedge invlpg(t) \;\rightarrow\; y.inva = inva_\sigma^{NS(t)}$$

- invalidation of all translations of certain guest.

$$guest(t) \wedge flusht(t) \;\rightarrow\; y.vmid = vmid(c^{NS(t)})$$

- invalidation of all translations.

$$host(t) \wedge flusht(t) \;\rightarrow\; y = (drop, all)$$

The last case we have directly from the definition, i.e., there is nothing to show. For the invalidation address (first case) we argue as follows.

$$\begin{aligned} y.inva &= mmu_Y^t.inva && \text{(definition)} \\ &= inva_\sigma^{NS(t)} && \text{(lemma 71.4)} \end{aligned}$$

For the invalidated virtual machine ID we argue similarly.

$$\begin{aligned} y.vmid &= mmu_Y^t.inva.vm && \text{(interconnect)} \\ &= vmid(t) && \text{(interconnect)} \\ &= vmid(c^{NS(t)}) && \text{(lemma 70)} \end{aligned}$$

Once again, using auxiliary configuration c' s.t.

$$c'.Z = \begin{cases} \delta_{tlb}(c^{NS(t)}.tlb, y) & Z = tlb \\ c^{NS(t)}.Z & \text{otherwise,} \end{cases}$$

we complete the induction step for the TLB component exactly as above.

$$\begin{aligned} sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^t, c^{NS(t)}) &\rightarrow sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^{t+1}, c') && \text{(lemma 18)} \\ &\rightarrow sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^{t+1}, c^{NS(t+1)}) && \text{(lemma 17)} \end{aligned}$$

7.5 Verifying Guard Conditions

In this section we show that the guard conditions are respected in all ISA steps until $NS(t+1)$, i.e.,

$$\forall n < NS(t+1) : \,\Gamma(mc^n, s(n)).$$

Directly from the induction hypothesis we obtain:

$$\forall n < NS(t) : \,\Gamma(mc^n, s(n)).$$

In case no steps are performed in cycle t, by definition we have

$$NS(t+1) = NS(t)$$

and the claim follows. Otherwise, we split cases on whether step

$$n = NS(t)$$

is a TLB step (see Sect. 7.5.1) or a processor core step (see Sect. 7.5.2).

Before we proceed to the induction step, for the implementation registers of the nested MMUs we show the following: in case a walk extension step is performed by mmu_Y in cycle t, the implementation register $mmu_Y.w_X$ contains valid walks on both, guest and user TLB steps (see Sect. 5.2.1).

Lemma 72.

$$wext_G(mmu_Y^t) \;\rightarrow\; valid(c^{NS(t)}.tlb, mmu_Y^t.w_G) \tag{1}$$

$$wext_U(mmu_Y^t) \;\rightarrow\; valid(c^{NS(t)}.tlb, mmu_Y^t.w_U, mmu_Y^t.w_G) \tag{2}$$

Proof of lemma 72.1. Directly from the induction hypothesis we derive

$$\begin{aligned} sim_{tlb}(mmu_Y^t, c^{NS(t)}) &\rightarrow sim_W(mmu_Y^t, c^{NS(t)}) && \text{(definition)} \\ &\rightarrow w_G(mmu_Y^t) \subseteq c^{NS(t)}.tlb && \text{(definition)} \\ &\rightarrow mmu_Y^t.w_G \in c^{NS(t)}.tlb && \text{(definition).} \end{aligned}$$

From the construction of the nested MMU (see Sect. 4.3) we derive that the walk extended in the guest walk register of mmu_Y is neither faulty nor complete.

$$\begin{aligned} wext_G(mmu_Y^t) &\rightarrow mmu_Y^t.fetch\text{-}pte_G && \text{(construction)} \\ &\rightarrow /f(mmu_Y^t.w_G) \wedge /mmu_Y^t.w_G.\ell[0] && \text{(invariant 3.2)} \end{aligned}$$ □

Proof of lemma 72.2. Again, from the induction hypothesis we derive

$$\begin{aligned} sim_{tlb}(mmu_Y^t, c^{NS(t)}) &\rightarrow sim_W(mmu_Y^t, c^{NS(t)}) && \text{(definition)} \\ &\rightarrow \textstyle\bigcup_X w_X(mmu_Y^t) \subseteq c^{NS(t)}.tlb && \text{(definition)} \\ &\rightarrow \textstyle\bigwedge_X mmu_Y^t.w_X \in c^{NS(t)}.tlb && \text{(construction).} \end{aligned}$$

From the construction of the nested MMU, we derive that the walks in the walk registers of mmu_Y are neither faulty nor complete.

$$\begin{aligned} wext_U(mmu_Y^t) &\rightarrow mmu_Y^t.fetch\text{-}pte_U && \text{(construction)} \\ &\rightarrow \textstyle\bigwedge_X /f(mmu_Y^t.w_X) \wedge /mmu_Y^t.w_X.\ell[0] && \text{(invariant 3.2)} \end{aligned}$$

From the arguments above we conclude

$$\textstyle\bigwedge_X valid(c^{NS(t)}.tlb, mmu_Y^t.w_X)$$

and it remains to show that i) the walks in the walk registers of mmu_Y are matching

$$mmu_Y^t.w_U \overset{\circ}{=} mmu_Y^t.w_G$$

and ii) the composition of these walks is not faulty.

$$/f(mmu_Y^t.w_U, mmu_Y^t.w_G)$$

Both statements follow from part (3) of invariant 3 analogous to the arguments above. □

7.5.1 TLB Steps

Assume that step n is generated by mmu_Y.

$$s(n) \in \Sigma_{tlb} \wedge tstep_Y(t)$$

First we consider execution at the level of guest, then — at the level of user.

i) From the induction hypothesis we have

$$guest(c^n).$$

From the interconnect and construction of the nested MMU we derive:

$$\begin{aligned} guest(t) &\rightarrow mmu_Y^t.upa \notin A_U && \text{(interconnect)} \\ &\rightarrow /treq_U(mmu_Y^t) && \text{(definition)} \\ &\rightarrow /tstep_{Y\ U}(t) && \text{(definition).} \end{aligned}$$

Thus, in case mmu_Y performs initialization of a guest walk ($winit_G(mmu_Y^t)$), for the stepping inputs by definition we have

$$s(n).(t,l) = (winit, guest)$$

which gives

$$T_G(c^n, s(n)).$$

In case mmu_Y performs extension of a guest walk ($wext_G(mmu_Y^t)$), for the stepping inputs by definition we have

$$s(n) = (wext, mmu_Y^t.w_G, \perp)$$

whereas from the first part of lemma 72 we have

$$valid(c^n.tlb, mmu_Y^t.w_G).$$

From the arguments above we again conclude

$$T_G(c^n, s(n)).$$

ii) From the induction hypothesis we have

$$user(c^n).$$

In case mmu_Y performs initialization or extension of a guest walk the arguments are identical to those presented in case i).

In case mmu_Y performs initialization of a user walk ($winit_U(mmu_Y^t)$), for the stepping inputs by definition we have

$$s(n).(t,l) = (winit, user)$$

which gives

$$T_U(c^n, s(n)).$$

In case mmu_Y performs extension of a user walk ($wext_U(mmu_Y^t)$), for the stepping inputs by definition we have

$$s(n) = (wext, mmu_Y^t.w_U, mmu_Y^t.w_G)$$

whereas from the second part of lemma 72 we have

$$valid(c^n.tlb, mmu_Y^t.w_U, mmu_Y^t.w_U).$$

From the arguments above we again conclude

$$T_U(c^n, s(n)).$$

7.5.2 Processor Core Steps

Assume that step n is performed by the processor core.

$$s(n) \in \Sigma_{core}.$$

The stepping used in [LOP] differs from the one given above in Sect. 7.1.4, however can be easily obtained from the induction hypothesis.

Lemma 73. *On steps of the processor core ($cstep(t)$) the following holds:*

$$used(w_Y)_\sigma^{i(t)} \rightarrow s(NS(t)).w_Y = h^t.w_Y$$

Before we proceed, we establish two more auxiliary results.

$$used(w_I)_\sigma^{i(t)} \wedge cstep(t) \rightarrow IF \leq cs(t) \tag{32}$$
$$used(w_E)_\sigma^{i(t)} \wedge cstep(t) \rightarrow EX \leq cs(t) \tag{33}$$

Proof of equation 32.

$$\begin{aligned} cstep(t) \wedge IF \not\leq cs(t) &\rightarrow cstep(t) \wedge jisr(t) \qquad \text{(definition)} \\ &\rightarrow \bigvee\nolimits_{k \leq \max \mathcal{J}(cs(t))} mca(t)[k] \qquad \text{(definition)} \\ &\rightarrow \exists k \leq \max \mathcal{J}(cs(t)) : \; mca_\sigma^{i(t)}[k] = 1 \qquad \text{(lemma 67)} \\ &\rightarrow il_\sigma^{i(t)} \leq 2 \qquad \text{(definition)} \\ &\rightarrow /used(w_I)_\sigma^{i(t)} \qquad \text{(definition)} \end{aligned}$$ □

The proof of equation 33 repeats the proof of equation 32, and therefore is omitted.

Proof of lemma 73. Using equations 32–33, for walk w_Y we derive:

$$\begin{aligned} s(NS(t)).w_Y &= w_Y{}_\sigma^{i(t)} \qquad \text{(lemma 64)} \\ &= h^t.w_Y \qquad \text{(IH).} \end{aligned}$$ □

That the walks coming from the walk registers satisfy the guard conditions of the specification was reflected in the induction hypothesis above. In case the first walk passed as the machine's input is used (for translation of the instruction address)

$$used(w_I)_\sigma^n$$

it is contained in the corresponding TLB

$$\begin{aligned} h^t.w_I &\in \begin{cases} c^{i(t)}.tlb^\circ & user(t) \\ c^{i(t)}.tlb & guest(t) \end{cases} \qquad \text{(invariant 6)} \\ &\in \begin{cases} c^n.tlb^\circ & user(c^n) \\ c^n.tlb & guest(c^n) \end{cases} \qquad \text{(lemma 64; IH)} \end{aligned}$$

and matches the corresponding translation request (by invariant 7 and lemma 64).

$$match(trq_I{}_\sigma^{i(c)}, h^t.w_I) \leftrightarrow match(trq_I{}_\sigma^n, h^t.w_I)$$

Applying lemma 73 (first part) we conclude

$$\Phi_I(c^n, s(n)).$$

Analogous to the above, in case the second walk passed to the specification machine is used (for translation of the effective address)

$$used(w_E)^n_\sigma$$

it is contained in the corresponding TLB

$$\begin{aligned} w_E.5^{q,t} &\in \begin{cases} c^{i(t)}.tlb^\circ & user(t) \\ c^{i(t)}.tlb & guest(t) \end{cases} && \text{(invariant 6)} \\ &\in \begin{cases} c^n.tlb^\circ & user(c^n) \\ c^n.tlb & guest(c^n) \end{cases} && \text{(lemma 64; IH)} \end{aligned}$$

and matches the corresponding translation request (by invariant 7 and lemma 64).

$$match(trq^{i(c)}_{E\ \sigma}, h^t.w_E) \leftrightarrow match(trq^n_{E\ \sigma}, h^t.w_E)$$

Finally, we apply lemma 73 (second part) and conclude

$$\Phi_E(c^n, s(n)).$$

This completes the induction step, and therefore the entire correctness proof for our implementation of the sequential machine with nested address translation. In the next chapter we show correctness of the NAT implementation in the pipelined multi-core machine.

Part IV

Multi-Core MIPS with NAT

8

Pipelined Processor with Nested MMUs

In Chap. 6 we presented the sequential implementation of the ISA specification from Sect. 3.3. In this chapter we proceed to pipeline the processor core from the latter implementation. Thus, in Sect. 8.1 we present the hardware mechanisms essential to implement the pipelined processor. Then, in Sect. 8.2 we collect some auxiliary machinery necessary to formally argue about the pipelined designs. Section 8.3 is a counterpart of Sect. 6.2, where we connect the pipelined processor to the cache memory system. Finally, in Sect. 8.4 we derive several crucial results on liveness of the pipelined implementation required later in Chap. 9.

8.1 Pipelined Processor

The implementation presented in this section assumes the absence of external interrupts, which according to [Sch13a] are provided by the advanced programmable interrupt controllers (APICs). Note, in order to integrate the APICs into the pipelined multi-core processor from Chap. 9, one should connect the external interrupt signals to the corresponding outputs of the APICs on every processor core.

8.1.1 Stall Engine Summary

In order to control execution of instruction in the pipelined machine we incorporate the stall-rollback engine developed in [LOP] from the original stall engine used in [KMP14]. Construction of the new control mechanism between stage $k-1$ and k of the pipeline is given in Fig. 33. As depicted in the figure, the original stall engine (on the left) is augmented with an additional circuitry and register within every stage. The original definition therefore can be rewritten as follows.

$$\begin{aligned} stall_k &= full_{k-1} \wedge (haz_k \vee stall_{k+1}) \\ ue_k &= full_{k-1} \wedge /stall_k \wedge /(rbp_{k-1} \vee rbr_k) \\ full_k^{t+1} &= stall_{k+1}^t \wedge /rollback_k^t \vee ue_k^t \end{aligned}$$

According to the definition above, the newly added hardware (on the right) is used i) to prevent updates and ii) to clear the pipeline stages. Definitions are collected below.

$$\begin{aligned} rbr_k &= misspec_k \vee rbr_{k+1} \\ rollback_{k-1} &= (rbp_{k-1} \vee rbr_k) \wedge /rhaz_k \\ rbp_{k-1}^{t+1} &= (rbp_{k-1}^t \vee rbr_k^t) \wedge rhaz_k^t \end{aligned}$$

For all stages, registers *rbp* are initialized with zeros on reset, just like the full bits. For the write back stage ($k=7$) we override the definitions and use

$$\begin{aligned} stall_7 &= 0 \\ rbr_7 &= 0. \end{aligned}$$

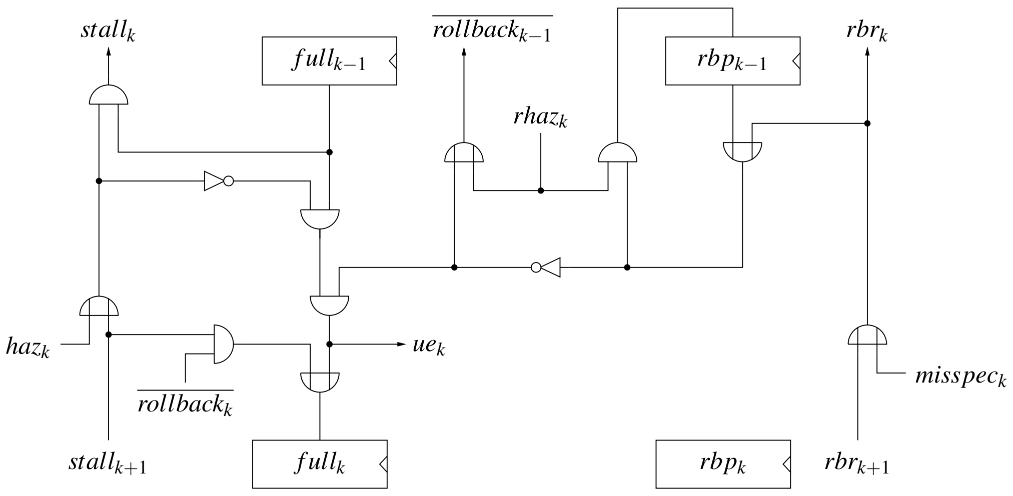


Fig. 33: Stall-rollback engine hardware between stages $k-1$ and k of the pipeline

Thus, whenever hardware detects that a certain pipeline stage does not operate on the relevant data, i.e., the *misspeculation* is detected in one of the stages, all stages above in the pipeline must be cleared. In the process of development of [LOP] it turned out that changes to the stall engine are unavoidable. Modified construction, called the *stall-rollback engine*, was proposed by Jonas Oberhauser. As depicted in Fig. 33, the stalling mechanism of [KMP14] has undergone significant changes.

Signal *misspec* raised in stage k propagates through the pipeline, triggering activation of the rollback request signals in all stages starting from k. Once stage k is ready for rollback (for instance, when the memory busy signal becomes low in stage k)[1], the full bit of stage $k-1$ is cleared (unless stage $k-1$ is updated in the same cycle). The latter is resp. signalled by the *rollback hazard* inactive for stage k and achieved by activation of the *rollback signal* for stage $k-1$, as depicted. Otherwise, in case stage k is not yet ready for rollback, the *rollback-pending bit* is set for stage $k-1$ resp. the full bit is *not* cleared. The latter rollback-pending bit effectively acts as a rollback request signal for stage k *while* the rollback hazard for this stage is high.

Stages stabilized by rollback-pending bits propagate stall signals in the same way as before. As a necessary operating condition we require that the rollback hazard in stage k is active only if stage above k is full and there is an active hazard signal in stage k.

$$rhaz_k \rightarrow full_{k-1} \wedge haz_k \tag{34}$$

Construction of the stall-rollback engine in full detail can be found in [LOP]. For convenience we introduce a very useful shorthand

$$rfull_k = full_k \wedge /rbp_k$$

for the *real full bit* at stage k, and equivalently rewrite the definitions of the update enable signals

$$\begin{aligned} ue_k &= full_{k-1} \wedge /stall_k \wedge /rbp_{k-1} \wedge /rbr_k \\ &= rfull_{k-1} \wedge /stall_k \wedge /rbr_k \end{aligned} \tag{35}$$

and the rollback-pending bits.

[1] Whenever the memory busy signal is low, according to the operating conditions for the cache memory system [KMP14], the memory input signals are not required to be stable. Therefore, the stage providing the latter input signals can be rolled-back.

$$\begin{aligned} rbp_{k-1}^{t+1} &= (rbp_{k-1}^t \vee rbr_k^t) \wedge rhaz_k^t \\ &= (rbp_{k-1}^t \vee rbr_k^t) \wedge (rhaz_k^t \vee /(rbp_{k-1}^t \vee rbr_k^t)) \\ &= (rbp_{k-1}^t \vee rbr_k^t) \wedge /rollback_{k-1}^t \end{aligned} \tag{36}$$

In the following lemmas we capture some of the important properties of the new control mechanism.

Lemma 74.

1. *Stall signals prevent stages above from update:*

$$stall_k \;\rightarrow\; /ue_{k-1}$$

2. *Rollback hazards prevent stages above from update:*

$$rhaz_k \;\rightarrow\; /ue_{k-1}$$

3. *Pending rollbacks are set only at full stages:*

$$rbp_k \;\rightarrow\; full_k$$

Proof of lemma 74.1. In case the stage above $k-1$ has no full bit, the claim follows directly from the definition.

$$/full_{k-2} \;\rightarrow\; /ue_{k-1}$$

Otherwise, using the definition of the stall signal we argue

$$full_{k-2} \wedge stall_k \;\rightarrow\; stall_{k-1}$$

and the claim follows again directly from the definition.

$$stall_{k-1} \;\rightarrow\; /ue_{k-1}$$ □

Proof of lemma 74.2. From the operating condition of the stall engine (equation 34) we conclude

$$rhaz_k \;\rightarrow\; full_{k-1} \wedge haz_k.$$

The latter by definition implies that the stall signal in stage k is active

$$full_{k-1} \wedge haz_k \;\rightarrow\; stall_k$$

and the claim follows by the first part (of lemma 74). □

Proof of lemma 74.3. By induction on the number of hardware cycles t. For the base case ($t = 0$) there is nothing to show, since after the hardware reset we clearly have

$$rbp_k^0 = 0.$$

For the induction step from t to $t+1$ we argue as follows. The rollback-pending bits are set only in presence of the rollback hazard signals.

$$rbp_k^{t+1} \;\rightarrow\; rhaz_{k+1}^t$$

Repeating the proof lines for the second part (of lemma 74) we derive that stage $k+1$ is stalled in cycle t.

$$rhaz_{k+1}^t \;\rightarrow\; stall_{k+1}^t$$

Moreover, stage k is not rolled-back in presence of the rollback hazard in cycle t

$$rhaz_{k+1}^t \;\rightarrow\; /rollback_k^t$$

which by definition completes the induction step.

$$stall_{k+1}^t \wedge /rollback_k^t \;\rightarrow\; full_k^{t+1}$$ □

Lemma 75.

1. *Updates of stages set the real full bits:*
$$ue_k^t \rightarrow rfull_k^{t+1} = 1.$$
2. *Rollback requests clear the real full bits:*
$$rbr_{k+1}^t \rightarrow rfull_k^{t+1} = 0.$$
3. *Truly full stages are not overwritten:*
$$rfull_k^t \wedge ue_k^t \rightarrow ue_{k+1}^t.$$
4. *Stages become truly full only after updates:*
$$/rfull_k^t \wedge rfull_k^{t+1} \rightarrow ue_k^t.$$
5. *Stages become empty only after updates or rollback requests from stages below:*
$$rfull_k^t \wedge /rfull_k^{t+1} \rightarrow ue_{k+1}^t \vee rbr_{k+1}^t.$$
6. *Unless a stage is updated, update of a stage below clears the real full bit:*
$$/ue_k^t \wedge ue_{k+1}^t \rightarrow rfull_k^{t+1} = 0.$$

Proof of lemma 75.1. For stage k updated in cycle t by definition we have
$$ue_k^t \rightarrow full_k^{t+1}.$$
From the second part of lemma 74 we also have
$$ue_k^t \rightarrow /rhaz_{k+1}^t.$$
We conclude that stage k has no rollback-pending bit in cycle $t+1$
$$/rhaz_{k+1}^t \rightarrow /rbp_k^{t+1}$$
and the claim follows. □

Proof of lemma 75.2. In case the rollback hazard signal for stage $k+1$ is active in cycle t, the claim follows directly from the definition.
$$rhaz_{k+1}^t \wedge rbr_{k+1}^t \rightarrow rbp_k^{t+1}$$
Otherwise, using the definition of the rollback signal we argue
$$/rhaz_{k+1}^t \wedge rbr_{k+1}^t \rightarrow rollback_k^t,$$
and for the update enable signal of stage k in cycle t we derive
$$rbr_{k+1}^t \rightarrow rbr_k^t \rightarrow /ue_k^t.$$
The claim follows by definition from the arguments above.
$$rollback_k^t \wedge /ue_k^t \rightarrow /full_k^{t+1}$$ □

Proof of lemma 75.3. By contradiction, assume
$$ue_{k+1}^t = 0.$$
From the definition of the update enable signal we conclude the following.

$$/rfull_k^t \vee stall_{k+1}^t \vee rbr_{k+1}^t$$

From the assumptions we know that stage k has a real full bit in cycle t. From the definition of the stall signal we derive

$$stall_{k+1}^t \wedge rfull_k^t \rightarrow stall_k^t$$

which contradicts the assumptions, since by definition of the update enable signal we have

$$stall_k^t \rightarrow /ue_k^t.$$

Finally, in case the rollback request signal for stage $k+1$ is active in cycle t, we derive

$$rbr_{k+1}^t \rightarrow rbr_k^t \rightarrow /ue_k^t$$

which gives a contradiction, and therefore completes the proof. □

Proof of lemma 75.4. From the definition of the real full bit we have the following.

$$/full_k^t \vee rbp_k^t$$

In case stage k is not full in cycle t, from the definition of the stall signal we have

$$/full_k^t \rightarrow /stall_{k+1}^t.$$

Using the definition of the full bits we conclude

$$/stall_{k+1}^t \wedge /ue_k^t \rightarrow /full_k^{t+1}$$

and the claim follows. Otherwise, if the rollback-pending bit for stage k is set in cycle t, we argue as follows. In case the rollback signal for stage k is active in cycle t, using the definition of the full bits we derive

$$rollback_k^t \wedge /ue_k^t \rightarrow /full_k^{t+1}.$$

Finally, in case the rollback request signal for stage k is inactive in cycle t, using the definition of the rollback-pending bits we obtain

$$rbp_k^t \wedge /rollback_k^t \rightarrow rbp_k^{t+1}.$$ □

Proof of lemma 75.5. From the definitions of the update enable (equation 35) and rollback signals we resp. have

$$rfull_k^t \wedge /rbr_{k+1}^t \wedge /ue_{k+1}^t \rightarrow stall_{k+1}^t$$

and

$$/rbp_k^t \wedge /rbr_{k+1}^t \rightarrow /rollback_k^t.$$

From the definitions of the full and the rollback-pending bits we resp. have

$$stall_{k+1}^t \wedge /rollback_k^t \rightarrow full_k^{t+1}$$

and

$$/rbp_k^t \wedge /rbr_{k+1}^t \rightarrow /rbp_k^{t+1}.$$ □

Proof of lemma 75.6. From the definition of the update enable signal we have

$$ue_{k+1}^t \rightarrow /stall_{k+1}^t.$$

Using the definition of the full bits we derive

$$/stall_{k+1}^t \wedge /ue_k^t \rightarrow /full_k^{t+1}$$

and the claim follows. □

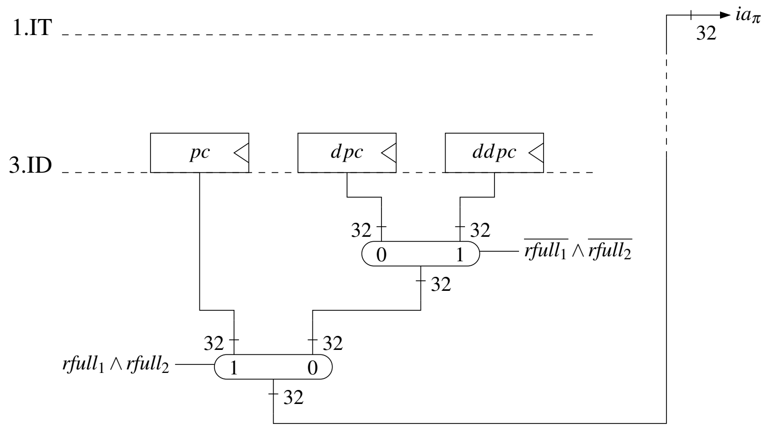


Fig. 34: Instruction address computation

8.1.2 Instruction Address

The instruction address in the seven-stage pipelined machine is taken from the *pc*, *dpc*, or *ddpc*, depending on the number of real full stages above the ID stage. Thus, in case all there are no real full stages, the instruction address is taken from the *ddpc*. In case only one stage has a real full bit, we use the *dpc* to fetch an instruction. Finally, if both stages have real full bits, we use the *pc*. Formally, we specify the instruction address follows.

$$ia_\pi = \begin{cases} ddpc_\pi & \overline{rfull_1} \wedge \overline{rfull_2} \\ dpc_\pi & rfull_1 \oplus rfull_2 \\ pc_\pi & rfull_1 \wedge rfull_2 \end{cases}$$

The implementation (as shown in Fig. 34) is trivial.

8.1.3 Interrupt Cause Pipeline

The internal events are collected gradually, throughout the instruction execution, within stages 1–5. The external event signals are collected in stage 6. We use the following shorthands to group the event signals collected in the various stages.

$$\begin{aligned} ev(1) &= 0_6 \circ gff \circ pff \circ malf \circ 0_2 \\ ev(2) &= 0_{11} \\ ev(3) &= 0_4 \circ sysc \circ ill \circ 0_5 \\ ev(4) &= 0_2 \circ malm \circ ovf \circ 0_7 \\ ev(5) &= gfm \circ pfm \circ 0_9 \\ ev(6) &= 0_9 \circ e \circ reset \end{aligned}$$

The first five of the signals above are fed into the registers of the cause pipeline.

$$\begin{aligned} ca.1_\pi.in &= ev(1) \\ k \in [2:5] \;\rightarrow\; ca.k_\pi.in &= ev(k) \vee ca.(k-1)_\pi \end{aligned}$$

Schematic construction of the cause pipeline is depicted in Fig. 35. Note, all stages of the cause pipeline consist of registers of the same size (11-bit wide), though in the "early" stages very few bits of these registers are actually in use. The latter can be easily fixed, but we do not bother in favor of a more simple implementation. For convenience, below we introduce short acronyms to refer to the particular bits of the cause pipeline registers in various stages. Naturally, the names are chosen to mimic the names of the corresponding event signals.

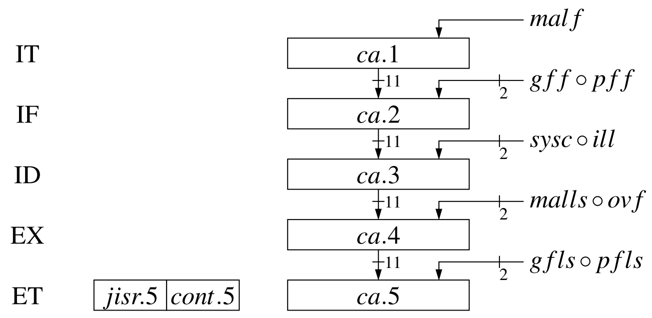


Fig. 35: Collecting event signals in the cause pipeline

index	10	9	8	7	6	5	4	3	2
acronym	`gm`	`pm`	`mm`	`of`	`sc`	`il`	`gf`	`pf`	`mf`

Also for convenience we introduce the following notation. For acronym x we define:

$$ca.k_\pi[>\mathrm{x}] \leftrightarrow \langle ca.k_\pi[\mathrm{x}:0]\rangle = 0.$$

Computation of most internal event signals remains straightforward: it precisely follows the specifications from Sect. 2.3. In contrast to the sequential implementation, the page faults and general-protection faults are explicitly lowered in the presence of other interrupts of lower priority.[2] For instance, the page faults on fetch and memory operation resp. are computed as follows.

$$\begin{aligned} pff &= /host \wedge ca.1_\pi.in[>\mathtt{mf}] \wedge f(mmu_I.wout) \\ pfm &= /host \wedge ca.5_\pi.in[>\mathtt{mm}] \wedge f(mmu_E.wout) \wedge mop.4_\pi \end{aligned}$$

Computation of the masked cause signals is performed in accordance with the ISA:

$$\begin{aligned} mca(1) &= ev(1) \\ k \in [2:5] \rightarrow mca(k) &= ev(k) \vee ca.(k-1)_\pi \\ mca(6) &= (ev(6) \vee ca.5_\pi) \wedge imask \end{aligned}$$

where the interrupt mask above is obtained from the status register.

$$imask = 1^9 \circ sr_\pi[1] \circ 1$$

Easy to see that computation of the page faults signals above can be equivalently rewritten as follows.

$$\begin{aligned} pff &= /host \wedge mca(1)[>\mathtt{mf}] \wedge f(mmu_I.wout) \\ pfm &= /host \wedge mca(5)[>\mathtt{mm}] \wedge f(mmu_E.wout) \wedge mop.4_\pi \end{aligned}$$

As depicted in Fig. 35, two more invisible registers are involved into the cause processing in stage 5 (apart from *ca*.5): *jisr*.5 and *cont*.5. Inputs of these registers are computed using the masked cause signal in stage 5.

$$\begin{aligned} jisr.5_\pi.in &\equiv mca(5) \neq 0_{11} \\ cont.5_\pi.in &\equiv f1(mca(5))[7:6] \neq 0_2 \end{aligned}$$

Note, the external event signals are ignored in the definitions above. Finally, using the latter registers, processor control signals *jisr* and *cont* are generated in the memory stage. The auxiliary control signal *exec* is defined exactly as in Chap. 6.

[2] Recall, in the sequential machine, execution of the current instruction is aborted once an unmasked interrupt of a non-continue type is discovered (see Sect. 6.1.2).

$$\begin{aligned} jisr &= ue_6 \wedge jisr.5_\pi \\ cont &= ue_6 \wedge cont.5_\pi \\ exec &\equiv jisr \to cont \end{aligned}$$

In the invariant below we justify early computation of the processor control signals.

Invariant 8. *In case stage 5 has a real full bit ($rfull_5$), the following holds:*

$$jisr.5_\pi \leftrightarrow ca.5_\pi \neq 0_{11} \tag{1}$$

$$cont.5_\pi \leftrightarrow f1(ca.5_\pi)[7:6] \neq 0_2 \tag{2}$$

$$mca(6) = ca.5_\pi \tag{3}$$

Proof of invariant 8.1. By induction on the number of hardware cycles t. For the base case ($t = 0$) there is nothing to show, since there are no full pipeline stages after reset ($/rfull_5^0$). For the induction step from t to $t+1$ we split cases on whether stage 5 is updated in cycle t.

- $ue_5^t = 1$. In case the stage is updated, we easily obtain the desired result.

$$\begin{aligned} jisr.5_\pi^{t+1} &\leftrightarrow mca(5)^t \neq 0_{11} && \text{(interconnect)} \\ &\leftrightarrow ev(5)^t \vee ca.4_\pi^t \neq 0_{11} && \text{(definition)} \\ &\leftrightarrow ca.5_\pi^{t+1} \neq 0_{11} && \text{(interconnect)} \end{aligned}$$

- $ue_5^t = 0$. In this case we assume that stage 5 has a real full bit in cycle t ($rfull_5^t$), since otherwise there is nothing to show (part (4) of lemma 75).

$$/rfull_5^t \wedge /ue_5^t \to /rfull_5^{t+1}$$

  To complete the induction step we argue as follows.

$$\begin{aligned} jist.5_\pi^{t+1} &= jisr.5_\pi^t && \text{(specification)} \\ &\leftrightarrow ca.5_\pi^t \neq 0_{11} && \text{(induction hypothesis)} \\ &\leftrightarrow ca.5_\pi^{t+1} \neq 0_{11} && \text{(specification)} \end{aligned}$$

□

Proof of the second part of invariant 8 is analogous to the proof above, whereas proof of the third part is trivial, given that the external event signals are off ($eev = 0_2$) in the scope of this chapter.

8.1.4 Ghost Pipeline for Translations in Use

In order to keep track of translations used by the pipeline machine in the course of instruction execution, we introduce a ghost pipeline of walks, outputs of the MMUs. For mmu_I we introduce ghost registers

$$w_I.k$$

in stages $k \in [1:5]$; for mmu_E — ghost register

$$w_E.5.$$

Obviously, we connect

$$\begin{aligned} w_I.1.in &= mmu_I.wout \\ w_E.5.in &= mmu_E.wout \end{aligned}$$

and for $k \in [2:5]$:

$$w_I.k.in = w_I.(k-1).$$

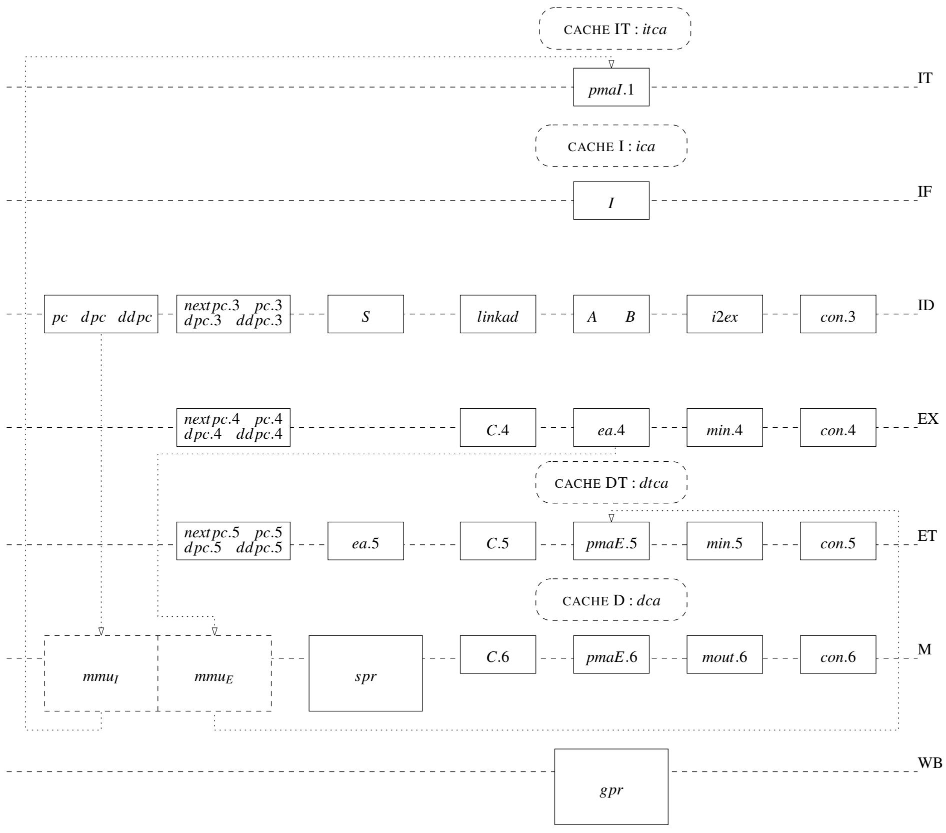


Fig. 36: Pipelined MIPS processor connected to four caches and two (nested) MMUs

8.1.5 Connecting Components

In this chapter we consider a pipelined processor connected to a sequentially consistent cache memory system (as constructed in [KMP14]) with four caches: two caches for translation of the instruction and effective addresses and two resp. for the instruction fetch and the data access. For simplicity we use the following intuitive names to abbreviate the caches in use.

$$\begin{aligned} itca_\pi &= h_\pi.ca(0) \\ ica_\pi &= h_\pi.ca(1) \\ dtca_\pi &= h_\pi.ca(2) \\ dca_\pi &= h_\pi.ca(3) \end{aligned}$$

As depicted in Fig. 36, we represent the cache for translation of instruction addresses by $itca_\pi$, whereas the cache for translation of effective addresses by $dtca_\pi$. Also as depicted in Fig. 36, two nested MMUs are connected in the memory stage. This subsection is spent to interconnect all these components formally.

Interconnect of MMUs

First we connect two MMUs both to the processor core and to the pair of caches. The following inputs are identical for both memory management units.

$$\begin{aligned} mmu_Y.pto_G &= pto_\pi \\ mmu_Y.pto_U &= npto_\pi \\ mmu_Y.invlpg &= ue_6 \wedge exec \wedge invlpg.5_\pi \wedge \overline{user} \\ mmu_Y.vmflush &= ue_6 \wedge exec \wedge flusht.5_\pi \wedge guest \\ mmu_Y.flush &= ue_6 \wedge exec \wedge flusht.5_\pi \wedge host \end{aligned}$$

Translation requests on both MMUs are raised only at the levels of user or guest, in the absence of *jisr*, and only for really full input stages. Translation request on mmu_E stays low in case no memory operation is performed.

$$\begin{aligned} treq_I &= \overline{host} \wedge \overline{jisr} \wedge mca(1)[\texttt{>mf}] \wedge rfull_0 \\ treq_E &= \overline{host} \wedge \overline{jisr} \wedge mca(5)[\texttt{>mm}] \wedge rfull_4 \wedge mop.4_\pi \end{aligned}$$

Actually, in case of a rollback, the translation request will simply be lowered and no pending bit will be set for the corresponding translation stage (see the definition of rollback hazard signals on p. 185). The latter is possible because the nested MMUs can handle aborts (see Sect. 4.2.2). Translation requests are also lowered to perform an invalidation request.

$$mmu_Y.treq = treq_Y \wedge /inval(mmu_Y)$$

Thus, the invalidation requests have priority over the translation requests.[3] The remaining inputs of mmu_I are connected as follows.

$$\begin{aligned} mmu_I.upa &= asid_\pi \circ ia_\pi.pa \\ mmu_I.mout &= itca_\pi.pdout \\ mmu_I.mbusy &= itca_\pi.mbusy \end{aligned}$$

For mmu_E the respective inputs are connected similarly.

$$\begin{aligned} mmu_E.upa &= asid_\pi \circ ea.4_\pi.pa \\ mmu_E.mout &= dtca_\pi.pdout \\ mmu_E.mbusy &= dtca_\pi.mbusy \end{aligned}$$

Interconnect of Stall Engine

The ordinary hazard signals, haz_1 and haz_5 below, make sure that in translated mode the translation stages are not updated while the corresponding MMUs are requested. The remaining hazard signals do not change compared to [LOP].

$$\begin{aligned} haz_1 &= treq_I \wedge (mmu_I.busy \vee inval(mmu_I)) \vee drain \\ haz_2 &= ica_\pi.mbusy \\ haz_3 &= haz_A \vee haz_B \\ haz_5 &= treq_E \wedge (mmu_E.busy \vee inval(mmu_E)) \\ haz_6 &= dca_\pi.mbusy \end{aligned}$$

As the name suggests, the *drain* signal above is required to drain the pipeline *behind* the *eret* instructions. Since the *eret* instructions are not necessarily executed (illegal at the level of user), for stages $k \geq 2$ we introduce the *interrupt return* signals, which indicate the presence only of the “legal” *eret* instructions in the pipeline.

$$iret_k = \begin{cases} eret(3) \wedge mca(3)[\texttt{>il}] & k = 2 \\ eret.k_\pi \wedge ca.k_\pi[\texttt{>il}] & \text{otherwise} \end{cases}$$

Then the *drain* signal is defined simply as follows.

$$drain = \bigvee_{k \geq 2} rfull_k \wedge iret_k$$

In the absence of the rollback hazard signals for the translation stages, the rollback hazard signals for the rollback engine are exactly as introduced in [LOP].

[3] Recall, our simple construction of the nested MMU supports at most one request at a time. Later on for this reason the steps of address translation will never be performed in cycles in which the corresponding processor core executes an invalidating instruction.

$$rhaz_k = \begin{cases} ica_\pi.mbusy & k = 2 \\ 0 & \text{otherwise} \end{cases}$$

The following invariant clearly follows from the construction of the stall engine (Sect. 8.1.1).

Invariant 9.

$$rbp_k \rightarrow k = 1$$

The program counters are restored (from the corresponding exception registers) on activation of signal

$$pcres = rfull_2 \wedge iret_2.$$

No new misspeculation signals are introduced.

$$\begin{aligned} misspec_2 &= pcres \\ misspec_5 &= jisr \end{aligned}$$

The lemmas below follow directly from the machine's control mechanism.

Lemma 76.

$$rbr_k^t \wedge ue_{k+1}^t \rightarrow \forall j \in [1:k] : /rfull_j^{t+1}$$

Proof of lemma 76. For stages $j < k$ the claim follows simply by part (2) of lemma 75. For stage k we argue as follows. From the definitions of the update enable and stall signals we resp. have

$$rbr_k^t \rightarrow /ue_k^t$$

and

$$ue_{k+1}^t \rightarrow /stall_{k+1}^t.$$

The claim follows from the arguments above and definition of the full bits.

$$/stall_{k+1}^t \wedge /ue_k^t \rightarrow /full_k^{t+1}$$ □

As it might be clear from the definitions above, on *pcres* the real full bit of the IT stage is cleared. As the *eret* instruction progresses down through the pipeline, the *drain* signal stays active and effectively prevents the new instructions from entering the pipeline. As a result, there are no real full stages behind the legal *eret* instruction.

Lemma 77. *For stages $k > 2$ the following holds:*

1. $$rfull_k^t \wedge iret_k^t \rightarrow \forall j \in [1:k-1] : /rfull_j^t$$
2. $$ue_{k+1}^t \wedge iret_k^t \rightarrow \forall j \in [1:k] : /rfull_j^{t+1}$$

Proof of lemma 77.1. By induction on the number of hardware cycles t. For the induction base ($t = 0$) there is nothing to show, since after the hardware reset we clearly have

$$\forall k : /rfull_k^0.$$

For the induction step from t to $t+1$ we split cases on whether stage k is updated in cycle t or not:

- $ue_k^t = 1$. From the definition of the update enable signal we conclude

  $$rfull_{k-1}^t \wedge iret_{k-1}^t.$$

  For stage $k = 3$ by definition we further conclude

  $$pcres(t)$$

  and therefore rbr_2^t. The claim follows by lemma 76.

$$\forall j \in [1:2] : /rfull_j^{t+1}$$

For stages $k > 3$ by definition we conclude

$$drain(t)$$

and therefore $/ue_1^t$. From the induction hypothesis we have

$$\forall j \in [1:k-2] : /rfull_j^t$$

and therefore

$$\forall j \in [1:k-1] : /ue_j^t.$$

From the arguments above and part (4) of lemma 75 we derive

$$\forall j \in [1:k-2] : /rfull_j^{t+1}.$$

For stage $k-1$ we conclude the claim using part (6) of lemma 75.

$$rfull_{k-1}^{t+1} = 0$$

• $ue_k^t = 0$. From part(4) of lemma 75 we derive

$$rfull_k^t \wedge iret_k^t$$

and therefore, using the induction hypothesis, we conclude

$$\forall j \in [1:k-1] : /rfull_j^t.$$

Analogous to the case above (ue_k^t) we argue

$$\forall j \in [1:k-1] : /ue_j^t.$$

Again, from the arguments above and part (4) of lemma 75 we derive

$$\forall j \in [1:k-1] : /rfull_j^{t+1}. \qquad \square$$

Proof of lemma 77.2. From the definition of the update enable signal (equation 35) we have

$$ue_{k+1}^t \ \rightarrow \ rfull_k^t$$

and therefore using part (1) of lemma 77 we conclude

$$\forall j \in [1:k-1] : /rfull_j^t.$$

Repeating the arguments from the proof above (part (1) of lemma 77) we argue as follows. First, we argue that stages above $k+1$ are not updated in cycle t.

$$\forall j \in [1:k] : /ue_j^t$$

From the arguments above and part (4) of lemma 75 we conclude

$$\forall j \in [1:k-1] : /rfull_j^{t+1}.$$

For stage k using part (6) of lemma 75 we derive

$$rfull_k^{t+1} = 0. \qquad \square$$

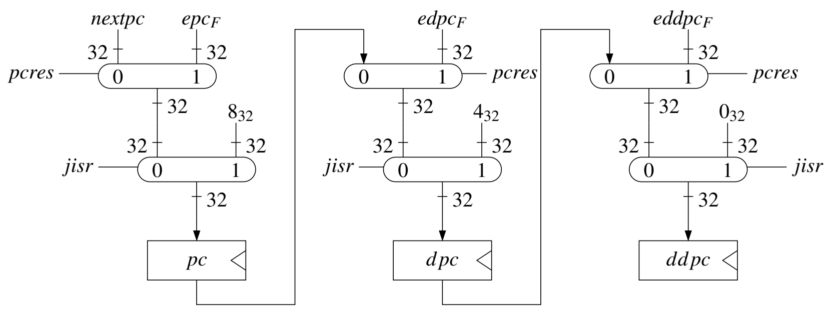


Fig. 37: PC environment

PC Environment

Computation of the next configuration PCs from [KMP14] is generalized in the following aspects. Apart of an additional circuitry added to support updates of the *ddpc*, there are extra multiplexers to restore all PCs from the corresponding exception registers on *pcres*. The implementation in Fig. 37 literally follows the specification from Sect. 2.4.3. Formally, for the pipelined machine we specify:

$$pc_\pi.in = \begin{cases} 8_{32} & jisr \\ epc_F & pcres \wedge /jisr \\ nextpc & \text{otherwise} \end{cases}$$

$$dpc_\pi.in = \begin{cases} 4_{32} & jisr \\ edpc_F & pcres \wedge /jisr \\ pc_\pi & \text{otherwise} \end{cases}$$

$$ddpc_\pi.in = \begin{cases} 0_{32} & jisr \\ eddpc_F & pcres \wedge /jisr \\ dpc_\pi & \text{otherwise} \end{cases}$$

where the *forwarded exception PCs* epc_F, $edpc_F$, and $eddpc_F$ are forwarded from the memory stage (see Sect. 8.1.6). Update of the program counters is performed on both *jisr* and *pcres*:

$$xpc.ce = ue_3 \vee jisr \vee pcres.$$

Note, there is no need to stabilize the PCs in our seven-stage pipeline. The instruction cache is accessed via the physical memory address register, which effectively acts as a latch for the instruction cache address in the five-stage pipeline from [LOP].

8.1.6 Forwarding Mechanism

Due to possible moves to the status registers pending in the local pipeline, we extend the forwarding mechanism from [KMP14] to deliver the most recent values of the SPR registers to the ID stage. Thus, we specify the *hit* signals to identify the data written to registers S and $Z \in \{epc, edpc, eddpc\}$ in pipeline stage $k \in [3:5]$:

$$\begin{aligned} hit_S[k] &= rfull_k \wedge sprw_\pi.k \wedge xad_\pi.k = rs_\pi \\ hit_Z[k] &= rfull_k \wedge sprw_\pi.k \wedge xad_\pi.k = \texttt{spr}[Z] \end{aligned}$$

Naturally, in order to forward the most recent data (in case there are several moves to the same register) we incorporate the *top* hit signals.

$$top_Z[k] = hit_Z[k] \wedge \bigwedge_{j<k} \overline{hit_Z[j]}$$

Using the notation above, we easily specify the input of the S register

$$S_\pi.in = \begin{cases} C.4_\pi.in & tops_S[3] \\ C.k_\pi & tops_S[k] \wedge k > 3 \\ spr_\pi(rs_\pi) & \text{otherwise} \end{cases}$$

and the forwarded exception PCs.

$$Z_F = \begin{cases} C.4_\pi.in & top_Z[3] \\ C.k_\pi & top_Z[k] \wedge k > 3 \\ Z_\pi & \text{otherwise} \end{cases}$$

8.2 Machinery for Description of Pipelines

In this small section we collect the machinery necessary to formally argue about the constructions presented in Sect. 8.1 above. Thus, the definitions of the execution mode and scheduling functions are given in Sect. 8.2.1 and 8.2.2 resp. In Sect. 8.2.3 we introduce the notion of *live circuit stages*, which turns out to be crucial to prove correctness of the pipelined implementation in Chap. 9.

8.2.1 Execution Modes

Execution mode of the pipeline is determined exactly as for the sequential machine, by values of the least significant bits of the special purpose registers $mode_\pi$ and $nmode_\pi$. We define the execution mode of the pipeline in cycle t as

$$mode(t) = nmode^t_\pi[0] \circ mode^t_\pi[0].$$

Thus, in our simple construction the execution mode is defined for the entire pipeline, i.e., in cycle t almost all instructions are executed at the same level of privilege. The only exception is a cycle immediately after *jisr*, in which an instruction in the write-back stage can have a different (higher) level of privilege. In the latter case, the instruction in the write-back stage finishes execution and leaves the pipeline in the same cycle (i.e., the cycle after *jisr*).

As before, we recognize only three modes of execution, which we intuitively encode by using the names: `user`, `guest`, and `host`.

$$mode(t) \in \{\texttt{user}, \texttt{guest}, \texttt{host}\}$$

Of course we order these three modes by the level of privilege as follows.

$$\texttt{host} < \texttt{guest} < \texttt{user}$$

For convenience we introduce the following shorthands.

$$\begin{aligned} user(t) &\equiv mode(t) = \texttt{user} \\ guest(t) &\equiv mode(t) = \texttt{guest} \\ host(t) &\equiv mode(t) = \texttt{host} \end{aligned}$$

In the following lemma we formalize a simple property that the pipeline mode changes only after execution of *jisr* or *eret*. Namely, the pipeline mode decreases only after execution of *jisr*, and increases only after execution of *eret*.

Lemma 78. *Assume that the pipeline mode changes in cycle t.*

$$mode(t) > mode(t+1) \rightarrow ue^t_6 \wedge jisr.5^t_\pi \tag{1}$$
$$mode(t) < mode(t+1) \rightarrow ue^t_6 \wedge \overline{jisr.5^t_\pi} \wedge eret.5^t_\pi \tag{2}$$

8.2.2 Scheduling Functions

In order to keep track of instructions executed in the various stages of the pipeline we incorporate the *scheduling functions*. In the pipeline with misspeculation and rollbacks we use the following definition, first introduced in Chap. 7 of [LOP]:

$$I(k,0) = 0$$

$$I(k,t+1) = \begin{cases} I(k,t)+1 & ue_k^t \wedge k = 1 \\ I(k-1,t) & ue_k^t \wedge k > 1 \\ I(R(t),t) & \overline{ue_k^t} \wedge rbr_k^t \\ I(k,t) & \text{otherwise} \end{cases}$$

where $R(t)$ denotes the maximal stage with an active rollback request signal, i.e.,

$$R(t) = \begin{cases} \max\{k \mid rbr_k^t\} & \exists k : rbr_k^t \\ 0 & \text{otherwise.} \end{cases}$$

In the following lemmas we capture the most important properties of the scheduling functions. Essentially, these lemmas are the counterparts to the corresponding lemmas from [KMP14].

Lemma 79. *Scheduling functions increase at real full stages.*

$$I(k-1,t) = I(k,t) + rfull_{k-1}^t$$

Proof of lemma 79. By induction on the number of hardware cycles t. For the induction base ($t = 0$) there is nothing to show, since after reset all real full bits are cleared

$$rfull_k^0 = 0$$

whereas for the scheduling functions by definition we have

$$I(k,0) = 0.$$

For the induction step from t to $t+1$ we argue as follows. First we proceed to show two auxiliary results which greatly simplify our proof. For $k > R(t)$ we claim that the following holds.

$$I(k,t+1) \;=\; I(k,t) + ue_k^t \tag{37}$$

$$(full \wedge /rbp)_{k-1}^{t+1} \leftrightarrow (full \wedge /rbp)_{k-1}^t \wedge stall_k^t \vee ue_{k-1}^t \tag{38}$$

Proof of equation 37. For stage $k = 1$ we obtain directly from the definition:

$$I(1,t+1) = I(1,t) + ue_1^t.$$

For stage $k > 1$ we split cases on whether stage k is updated in cycle t or not.

- $ue_k^t = 0$. In case stage k is not updated, we have

$$\begin{aligned} I(k,t+1) &= I(k,t) && \text{(definition)} \\ &= I(k,t) + ue_k^t && \text{(assumption).} \end{aligned}$$

- $ue_k^t = 1$. Otherwise, we argue as follows.

$$\begin{aligned} I(k,t+1) &= I(k-1,t) && \text{(definition)} \\ &= I(k,t) + rfull_{k-1}^t && \text{(induction hypothesis)} \\ &= I(k,t) + ue_k^t && \text{(assumption)} \end{aligned}$$

Note, from the construction of the stall engine (equation 35) we know

$$ue_k^t \;\rightarrow\; rfull_{k-1}^t.$$

□

Proof of equation 38.

($\rightarrow$) In order to show sufficiency, from the construction of the stall engine we derive.

$$\begin{aligned} full^{t+1}_{k-1} &\rightarrow stall^t_k \wedge /rollback^t_{k-1} \vee ue^t_{k-1} \\ /rollback^t_{k-1} \wedge /rbp^{t+1}_{k-1} &\rightarrow /rbp^t_{k-1} \end{aligned}$$

($\leftarrow$) For necessity, again from the construction of the stall engine, we argue as follows.

$$\begin{aligned} /rbr^t_k \wedge /rbp^t_{k-1} &\rightarrow /rbp^{t+1}_{k-1} \wedge /rollback^t_{k-1} \\ /rollback^t_{k-1} \wedge stall^t_k &\rightarrow full^{t+1}_{k-1} \end{aligned}$$

Recall, we prove the statement only for stages $k > R(t)$, which implies $/rbr^t_k$. In case stage above k is updated in cycle t (ue^t_{k-1}), the claim follows directly from lemma 75. □

Using the statements above we can easily complete the induction step. Below we split cases on whether stage k is rolled-back in cycle t or not.

- $rbr^t_k = 1$. In this case all stages above k are rolled-back as well (rbr^t_{k-1}). From lemma 75 we have
$$rfull^{t+1}_{k-1} = 0.$$
For the scheduling functions we derive:
$$\begin{aligned} I(k-1,t+1) &= I(R(t),t) \qquad \text{(definition)} \\ &= I(k,t+1) \qquad \text{(definition).} \end{aligned}$$

- $rbr^t_k = 0$. Otherwise, all stages below k are not rolled-back as well ($/rbr^t_{k+1}$). Using equation 38 we rewrite the claim
$$I(k-1,t+1) = I(k,t+1) + /rfull^t_{k-1} \wedge stall^t_k \vee ue^t_{k-1}.$$
and split cases further on whether stage k is stalled in cycle t or not. In the first case ($stall^t_k$), from the construction of the stall engine we derive
$$stall^t_k \rightarrow /ue^t_k \wedge /ue^t_{k-1}$$
and argue as follows.
$$\begin{aligned} I(k-1,t+1) &= I(k-1,t) \qquad \text{(equation 37)} \\ &= I(k,t) + rfull^t_{k-1} \qquad \text{(induction hypothesis)} \\ &= I(k,t+1) + rfull^t_{k-1} \qquad \text{(equation 37)} \end{aligned}$$
In the second case ($/stall^t_k$), again from the construction of the stall engine (equation 35) we derive
$$ue^t_k \leftrightarrow rfull^t_{k-1} \wedge /stall^t_k \wedge /rbr^t_k$$
which allows us to complete the induction step.
$$\begin{aligned} I(k-1,t+1) &= I(k-1,t) + ue^t_{k-1} \qquad \text{(equation 37)} \\ &= I(k,t) + rfull^t_{k-1} + ue^t_{k-1} \qquad \text{(induction hypothesis)} \\ &= I(k,t+1) + ue^t_{k-1} \qquad \text{(equation 37)} \end{aligned}$$
□

The following lemma gives the value of the scheduling functions of all stages in the next cycle.

Lemma 80. *In stages which are not rolled-back, scheduling functions increase with updates; in the remaining stages — scheduling functions are reset to $I(R(t),t)$.*

$$I(k,t+1) = \begin{cases} I(k,t) + ue^t_k & k > R(t) \\ I(R(t),t) & \textit{otherwise} \end{cases}$$

Proof of lemma 80. For stages $k > R(t)$ we refer the reader to the proof of equation 37. In the latter proof, all occasions of “induction hypothesis” should be simply replaced by “lemma 79”. For the remaining stages ($k \leq R(t)$) the claim follows by definition. Note, from the definition of the update enable signal we have

$$rbr_k^t \rightarrow /ue_k^t. \qquad \square$$

To streamline the proofs in the next sections we also formulate the following simple properties of the scheduling functions which could be easily derived formally.

Lemma 81. *For the scheduling functions of stages k and $\ell > k$ the following holds.*

$$I(k,t) = I(\ell,t) + \sum_{j=k}^{\ell-1} rfull_j^t$$

Lemma 82. *Assume that the memory stage is updated in cycle $t < t'$ (ue_6^t).*

$$I(6,t) < I(6,t')$$

8.2.3 Lowest Non-Live Stage

As we argue later in Sect. 9.6, to show correctness of the pipelined machine with misspeculation it suffices to consider only those instructions which are executed (in the memory stage) without rollbacks. We say that instruction in *circuit* stage k in cycle t is *live* if the following holds.

$$\begin{aligned} live(k,t) \equiv k = 7 \lor \exists t' \geq t : \; (I(6,t') = I(k,t) \land ue_6^{t'} \land \\ \forall \tilde{t} \in [t:t'] \; \forall \tilde{k} \in [k:6] : \; I(\tilde{k},\tilde{t}) = I(k,t) \rightarrow /rbr_{\tilde{k}}^{\tilde{t}} \;) \end{aligned}$$

Note, since instructions in the write-back stage are never rolled-back, for all cycles t we of course derive $live(7,t)$. The next lemma follows directly from the definition.

Lemma 83. *Assume $k' > k$.*

$$live(k,t) \rightarrow live(k',t)$$

Circuit stage $\mu(t)$ is the *lowest non-live stage* in cycle t:

$$\mu(t) = \max\{k \in [1:7] \mid /live(k,t)\}.$$

Note, in case all stages are live in cycle t

$$\forall k \in [1:7] : \; live(k,t)$$

we of course obtain

$$\mu(t) = \max \emptyset = 0.$$

First, we argue that *pipeline* stage $\mu(t) < 7$ has a real full bit in cycle t.

Lemma 84. *Assume $\mu(t) < 7$.*

$$rfull_{\mu(t)}^t$$

Proof of lemma 84. In case all stages are live in cycle t, there is nothing to show: technical stage zero is always truly full.

$$rfull_0^t$$

Otherwise, from the definition we have

$$/live(\mu(t),t) \land live(\mu(t)+1,t).$$

Clearly we have

$$I(\mu(t),t) \neq I(\mu(t)+1,t)$$

and the claim follows from lemma 79. $\square$

By definition the stage below $\mu(t)$ is live in cycle t. The simple lemma below follows.

Lemma 85. *Assume that the stage below $\mu(t)$ is updated in cycle t ($ue^t_{\mu(t)+1}$).*

$$live(\mu(t)+2, t+1)$$

Proof of lemma 85. In case stage $\mu(t)+2$ is updated in cycle t, by definition we have

$$I(\mu(t)+2, t+1) = I(\mu(t)+1, t).$$

Otherwise, from the construction of the stall engine (part (3) of lemma 75) we know that stage $\mu(t)+1$ does not have a real full bit in cycle t

$$ue^t_{\mu(t)+1} \wedge /ue^t_{\mu(t)+2} \;\rightarrow\; /rfull^t_{\mu(t)+1}$$

and easily derive

$$\begin{aligned} I(\mu(t)+2, t+1) &= I(\mu(t)+2, t) && \text{(definition)}\\ &= I(\mu(t)+1, t) && \text{(lemma 79).} \end{aligned}$$

From the definition of μ we clearly have

$$live(\mu(t)+1, t)$$

and therefore

$$live(\mu(t)+2, t+1).$$ □

Lemma 86. *Assume $\mu(t) \in [1:4]$ and the stage below $\mu(t)$ is updated in cycle t ($ue^t_{\mu(t)+1}$); assume the rollback request signal at stage $\mu(t)$ is inactive in cycle t ($/rbr^t_{\mu(t)}$).*

$$/live(\mu(t)+1, t+1)$$

Proof of lemma 86. By contradiction; assume

$$live(\mu(t)+1, t+1).$$

For the scheduling function by definition we have

$$I(\mu(t)+1, t+1) = I(\mu(t), t)$$

which together with the above implies

$$\begin{aligned} &\exists t' \geq t+1 :\; I(6, t') = I(\mu(t), t) \wedge ue^{t'}_6 \;\wedge \\ &\forall \tilde{t} \in [t+1 : t'] \;\forall \tilde{k} \in [\mu(t)+1 : 6] :\; I(\tilde{k}, \tilde{t}) = I(\mu(t), t) \rightarrow /rbr^{\tilde{t}}_{\tilde{k}}. \end{aligned} \tag{39}$$

From the assumption ($/rbr^t_{\mu(t)}$) for cycle t we by definition have

$$\forall \tilde{k} \in [\mu(t) : 6] : /rbr^t_{\tilde{k}} \tag{40}$$

and proceed to show the following.

$$\forall \tilde{t} \in [t+1 : t'] :\; I(\mu(t), \tilde{t}) = I(\mu(t), t) \rightarrow /rbr^{\tilde{t}}_{\mu(t)} \tag{41}$$

Proof of equation 41. By contradiction. For cycle $\tilde{t}$ we assume

$$I(\mu(t), \tilde{t}) = I(\mu(t), t) \wedge rbr^{\tilde{t}}_{\mu(t)}$$

and split cases on whether stage $\mu(t)$ has a real full bit in cycle $\tilde{t}$ or not:

- $rfull^{\tilde{t}}_{\mu(t)} = 0$. From lemma 79 we have

  $$I(\mu(t)+1, \tilde{t}) = I(\mu(t), t)$$

  which together with equation 39 we gives

  $$rbr^{\tilde{t}}_{\mu(t)+1} = 0.$$

  Using the definition of the *misspec* signals we derive a contradiction:

  $$/rfull^{\tilde{t}}_{\mu(t)} \wedge /rbr^{\tilde{t}}_{\mu(t)+1} \rightarrow /rbr^{\tilde{t}}_{\mu(t)}.$$

- $rfull^{\tilde{t}}_{\mu(t)} = 1$. In this case we argue as follows. Instruction

  $$I(\mu(t)+1, t+1) = I(\mu(t), t)$$

  is live (equation 39), and therefore can be found in stage $k \in [\mu(t)+1 : 6]$ in cycle $\tilde{t}$.

  $$I(k, \tilde{t}) = I(\mu(t), t)$$

  The contradiction follows.

  $$\begin{aligned} I(\mu(t), \tilde{t}) &> I(\mu(t)+1, \tilde{t}) \qquad \text{(lemma 79)} \\ &\geq I(k, t) \qquad \text{(lemma 81)} \\ &= I(\mu(t), \tilde{t}) \qquad \text{(assumption)} \end{aligned}$$ □

From the arguments above (equations 39–41) we conclude

$$live(\mu(t), t)$$

which is a contradiction by definition. □

8.3 Cache Memory System in Pipelined Processor

This is a counterpart of Sect. 6.2 for the pipelined machine. The arguments presented in Sects. 6.2.1–6.2.4 are mostly repeated in Sects. 8.3.1–8.3.4 resp. to fit the hardware description of the pipelined machine. In contrast to Chap. 6, in the end of this section we extract the sequence of processor accesses (A), crucial for the correctness proofs performed in Chap. 9.

8.3.1 Connections to Caches

Connections to the instruction ($itca_\pi$) and the data translation cache ($dtca_\pi$) are simple. Since in the scope of this thesis the MMUs only read the page tables

$$\begin{aligned} itca_\pi.(pr, pw, pcas) &= 100 \\ dtca_\pi.(pr, pw, pcas) &= 100 \end{aligned}$$

there is no need to provide inputs other than the processor address (in the data fields):

$$\begin{aligned} itca_\pi.pa &= mmu_I.ma.l \\ dtca_\pi.pa &= mmu_E.ma.l. \end{aligned}$$

The remaining inputs are the processor requests, which we connect as follows:

$$\begin{aligned} itca_\pi.preq &= mmu_I.mreq \\ dtca_\pi.preq &= mmu_E.mreq. \end{aligned}$$

Connections to the instruction (ica_π) and the data cache (dca_π) remain almost unchanged. The processor addresses are now connected to the physical memory address registers of the corresponding pipeline stages:

$$\begin{aligned} ica_\pi.pa &= pmaI.1_\pi.l \\ dca_\pi.pa &= pmaE.5_\pi.l. \end{aligned}$$

Since the instruction fetch and the memory stages shift down in the pipeline, the corresponding control registers are used to provide the access types

$$\begin{aligned} ica_\pi.(pr, pw, pcas) &= 100 \\ dca_\pi.(pr, pw, pcas) &= l.5_\pi \circ s.5_\pi \circ cas.5_\pi \end{aligned}$$

and activate the processor requests.

$$\begin{aligned} ica_\pi.preq &= full_1 \\ dca_\pi.preq &= full_5 \wedge mop.5_\pi \wedge /jisr.5_\pi \end{aligned}$$

Data for the memory access are obviously taken from the memory input stage (stage 5) as well. The processor compare-data are now connected directly to one of the SPR outputs (in contrast to [KMP14], where they were coming from the GPR).

$$\begin{aligned} dca_\pi.pbw &= bw.5_\pi \\ dca_\pi.pdin &= min.5_\pi \\ dca_\pi.pcdin &= cdata_\pi \end{aligned}$$

8.3.2 Stability of Inputs to Caches

In order to show that interconnect of the cache memory system from the section above meets the operating conditions of the cache memory system (see Sect. 6.2.2), we proceed to prove the counterpart of lemma 50 for the pipelined processor.

Lemma 87.

- *For any cache:*
$$h^t_\pi.ca(i).mbusy \;\rightarrow\; h^t_\pi.ca(i).preq \tag{42}$$
- *For the instruction translation cache:*
$$itca^t_\pi.mbusy \;\rightarrow\; mmu^{t+1}_I.(ma, mreq) = mmu^t_I.(ma, mreq) \tag{43}$$
- *For the instruction cache:*
$$ica^t_\pi.mbusy \;\rightarrow\; /ue^t_1 \wedge full^{t+1}_1 \tag{44}$$
- *For the data translation cache:*
$$dtca^t_\pi.mbusy \;\rightarrow\; mmu^{t+1}_E.(ma, mreq) = mmu^t_E.(ma, mreq)$$
- *For the data cache:*
$$dca^t_\pi.mbusy \;\rightarrow\; /ue^t_5 \wedge full^{t+1}_5$$

Proof of lemma 87. For cache i we argue by induction on the number of hardware cycles t. For the induction base ($t = 0$) there is nothing to show.

$$h^0_\pi.ca(i).mbusy = 0$$

For the induction step from t to $t+1$ we first show the remaining statements using the induction hypothesis (equation 42). Note, for the translation caches the arguments do not change compared to those presented in the proof of lemma 50, and therefore are omitted. For the instruction and data caches the arguments become more interesting.

Due to rollbacks below in the pipeline, we need to stabilize inputs to the instruction cache. For that purpose a new construction of the stall engine was elaborated (see Sect. 8.1.1). Now, an active rollback hazard signal for the instruction fetch stage ($rhaz_2$) protects the real full bit of the stage above, i.e., the request to the instruction cache, from clearing. Formally we derive:

$$\begin{aligned} ica^t_\pi.mbusy &\to ica^t_\pi.preq \wedge haz^t_2 \wedge rhaz^t_2 \qquad \text{(equation 42)} \\ &\to full^t_1 \wedge haz^t_2 \wedge rhaz^t_2 \qquad \text{(interconnect)} \\ &\to stall^t_2 \wedge stall^t_1 \wedge /rollback^t_1 \qquad \text{(definition)} \\ &\to /ue^t_1 \wedge full^{t+1}_1 \qquad \text{(definition).} \end{aligned}$$

For the data cache there is no need to stabilize the input stage since the memory stage is never rolled-back ($/rbr_6$). Again, formally we have:

$$\begin{aligned} dca^t_\pi.mbusy &\to dca^t_\pi.preq \wedge haz_6 \qquad \text{(equation 42)} \\ &\to full^t_5 \wedge haz^t_6 \qquad \text{(interconnect)} \\ &\to stall^t_6 \wedge (stall^t_5 \vee /full^t_4) \qquad \text{(definition)} \\ &\to /ue^t_5 \wedge full^{t+1}_5 \qquad \text{(definition).} \end{aligned}$$

To complete the induction step we argue as follows. We assume

$$h^{t+1}_\pi.ca(i).mbusy = 1,$$

since otherwise there is nothing to show. Next we split cases on whether cache i is busy in cycle t.

- In case cache i is busy
  $$h^t_\pi.ca(i).mbusy = 1,$$
  the claim follows directly from the lines above. For the instruction translation cache we derive:
  $$\begin{aligned} itca^t_\pi.mbusy &\to itca^t_\pi.preq \qquad \text{(equation 42)} \\ &\to mmu^t_I.mreq \qquad \text{(interconnect)} \\ &\to mmu^{t+1}_I.mreq \qquad \text{(equation 43)} \\ &\to itca^{t+1}_\pi.preq \qquad \text{(interconnect).} \end{aligned}$$
  In turn, for the instruction cache we simply have:
  $$\begin{aligned} ica^t_\pi.mbusy &\to full^{t+1}_1 \qquad \text{(equation 44)} \\ &\to ica^{t+1}_\pi.preq \qquad \text{(interconnect).} \end{aligned}$$
  The corresponding arguments for the data translation cache and the data cache are analogous, and therefore are omitted.
- Otherwise, in case cache i is not busy
  $$h^t_\pi.ca(i).mbusy = 0,$$
  one can literally follow the corresponding lines in the proof of lemma 50, and complete the induction step. □

8.3.3 Accesses of Hardware Computation

In this small section we identify the accesses occurring in the hardware computation for later reference. Similar to Sect. 6.2.3, we go through the interfaces of caches interconnected in

Sect. 8.3.1, and summarize on the connected signals for every port. Given that (i,k) is a non-flushing access ending in cycle t, according to [KMP14], we have:

$$\begin{aligned} acc(i,k).a &= h^t_\pi.ca(i).pa \\ acc(i,k).data &= h^t_\pi.ca(i).pdin \\ acc(i,k).cdata &= h^t_\pi.ca(i).pcdin \\ acc(i,k).bw &= h^t_\pi.ca(i).pbw \\ acc(i,k).type &= h^t_\pi.ca(i).(pr,pw,pcas,0). \end{aligned}$$

Below we instantiate $acc(i,k)$ for all ports of the single-core machine.

Accesses of Processor Core

For the instruction cache (ica_π) we have:

$$\begin{aligned} acc(1,k).a &= pmaI.1^t_\pi.l \\ acc(1,k).type &= 1000. \end{aligned}$$

For the data cache (dca_π) we have:

$$\begin{aligned} acc(3,k).a &= pmaE.5^t_\pi.l \\ acc(3,k).data &= min.5^t_\pi \\ acc(3,k).cdata &= cdata^t_\pi \\ acc(3,k).bw &= bw.5^t_\pi \\ acc(3,k).type &= l.5^t_\pi \circ s.5^t_\pi \circ cas.5^t_\pi \circ 0. \end{aligned}$$

Accesses of MMUs

For the access address, from the construction of the nested MMU we obtain

$$mmu^t_Y.ma = ptea(mmu^t_Y).$$

For the instruction translation ($itca_\pi$) we have:

$$\begin{aligned} acc(0,k).a &= ptea(mmu^t_I).l \\ acc(0,k).type &= 1000. \end{aligned}$$

For the data translation cache ($dtca_\pi$) we have:

$$\begin{aligned} acc(2,k).a &= ptea(mmu^t_E).l \\ acc(2,k).type &= 1000. \end{aligned}$$

8.3.4 Relating Endings of Accesses with Hardware Control Signals

Lemmas 88 and 89 below are the counterparts of lemmas [9.12] and [9.13] resp. from [KMP14] about accesses performed by the processor core. The latter two lemmas are reformulated to fit the pipelined processor utilizing four caches. Recall, every update of the instruction fetch stage is accompanied by a read access ending in the instruction cache and vise versa. The analogous result holds for the data cache and updates of the memory stage executing a memory operation.

Lemma 88.

$$\begin{aligned} ue^t_2 &\rightarrow \exists k : e(1,k) = t \wedge acc(1,k).r && (1) \\ exec(t) \wedge mop.5^t_\pi \wedge ue^t_6 &\rightarrow \exists k : e(3,k) = t \wedge \overline{acc(3,k).f} && (2) \end{aligned}$$

Proof of lemma 88.1. First, using the definitions of the update enable and the rollback hazard signals we derive

$$\begin{aligned} ue_2^t &\rightarrow full_1^t \wedge /rhaz_2^t \\ &\rightarrow /ica_\pi^t.mbusy. \end{aligned}$$

From the arguments above ($full_1^t$) and interconnect of the instruction cache (ica_π) we have

$$ica_\pi^t.preq = 1.$$

From the construction of caches [KMP14] we know that some non-flushing access to cache ica_π ends in cycle t

$$ica_\pi^t.preq \wedge /ica_\pi^t.mbusy \;\rightarrow\; someend(1,t) \wedge /flushend(1,t)$$

which by definition implies the claim, since for some k we conclude

$$e(1,k) = t \wedge /acc(1,k).f.$$ □

Proof of lemma 88.2. Using the definitions of the update enable and the stall signals we argue

$$\begin{aligned} ue_6^t &\rightarrow full_5^t \wedge /stall_6^t \\ &\rightarrow /haz_6^t \qquad (stall_7 = 0) \\ &\rightarrow /dca_\pi^t.mbusy \qquad \text{(definition)}. \end{aligned}$$

Next we proceed to show $/jisr.5_\pi^t$. By contradiction we argue as follows:

$$\begin{aligned} ue_6^t \wedge jisr.5_\pi^t &\rightarrow jisr(t) \\ mop.5_\pi^t \wedge jisr.5_\pi^t &\rightarrow /cont(t) \end{aligned}$$

which gives a contradiction, since by definition we derive $/exec(t)$. Therefore, from the arguments above ($full_5^t$ and $/jisr.5_\pi^t$) and interconnect of the data cache (dca_π) we have

$$dca_\pi^t.preq = 1.$$

Repeating the arguments presented in the proof of part (1) (of lemma 88) we derive

$$someend(3,t) \wedge /flushend(3,t)$$

and the claim follows, since for some k we conclude

$$e(3,k) = t \wedge /acc(3,k).f.$$ □

Lemma 89.

$$acc(1,k).r \wedge e(1,k) = t \;\rightarrow\; (\, /stall_3^t \wedge /rollback_1^t \rightarrow ue_2^t \,) \tag{1}$$

$$\overline{acc(3,k).f} \wedge e(3,k) = t \;\rightarrow\; exec(t) \wedge mop.5_\pi^t \wedge ue_6^t \tag{2}$$

Proof of lemma 89.1. From the assumptions and construction of caches [KMP14] we derive

$$/acc(1,k).f \wedge e(1,k) = t \;\rightarrow\; someend(1,t) \wedge /flushend(1,t)$$

which by definition gives

$$ica_\pi^t.preq \wedge /ica_\pi^t.mbusy.$$

From the arguments above and interconnect of the instruction cache (ica_π) we have

$$full_1^t \wedge /haz_2^t \wedge /rhaz_2^t$$

and using the assumptions we proceed as follows:

$$\begin{aligned} /stall_3^t \wedge /haz_2^t &\rightarrow /stall_2^t \\ /rollback_1^t \wedge /rhaz_2^t &\rightarrow /rbp_1^t \wedge /rbr_2^t. \end{aligned}$$

The latter completes the proof, since using the definition we conclude

$$full_1^t \wedge /stall_2^t \wedge /rbp_1^t \wedge /rbr_2^t \;\rightarrow\; ue_2^t.$$ □

Proof of lemma 89.2. Analogous to the proof of part (1) (of lemma 89) we argue

$$/acc(3,k).f \wedge e(3,k) = t \;\rightarrow\; someend(3,t) \wedge /flushend(3,t)$$

which by definition gives

$$dca_{\pi}^{t}.preq \wedge /dca_{\pi}^{t}.mbusy.$$

From the arguments above and interconnect of the data cache (dca_{π}) we have the following.

$$full_5^t \wedge mop.5_{\pi}^t \wedge /jisr.5_{\pi}^t \wedge /haz_6^t$$

Using the definitions we conclude

$$/stall_6^t \wedge /rbr_6^t.$$

Moreover, from invariant 9 we have $/rbp_5^t$, which by definition implies ue_6^t. Finally, from the arguments above ($/jisr.5_{\pi}^t$) using the definition we conclude $/jisr(t)$, and the claim follows. □

Note, accesses performed by the nested MMUs are covered by the corresponding lemmas from Sect. 6.2, which apply to the pipelined machine without changes.

8.3.5 Extracting Sequence of Processor Accesses

Following the approaches introduced in [KMP14] and [LOP] in this section we proceed to extract the sequence of accesses performed by processors in course of execution of memory operations. Accesses to the translation and data caches performed by the processors, or *processor accesses* for short, can be easily extracted. Note, accesses to the instruction caches and flushes are excluded.

$$A(t) = \{(i,k) \in E(t) \mid (i \bmod 4) \neq 1 \wedge /acc(i,k).f\}$$

For convenience we abbreviate the number of the processor accesses by

$$na(t) = \#A(t).$$

Of course we define the order of these accesses by introducing a dedicated function. For $y < na(t)$ we define:

$$\begin{aligned} x(0,t) &= \min \{n \mid NE(t)+n \in seq(A(t))\} \\ x(y,t) &= \min \{n \mid NE(t)+n \in seq(A(t)) \wedge n > x(y-1,t)\}. \end{aligned}$$

Using function x for indices

$$y \in [0 : na(t)-1]$$

we can easily extract from sequence acc' the *sequence of processor accesses* $xacc_t'$:

$$xacc_t'[y] = acc'[NE(t)+x(y,t)].$$

The sequence of memory configurations obtained by executing only the accesses from sequence $xacc_t'$ is defined as follows.

$$\begin{aligned} M_0 &= m(h_{\pi}^t) \\ M_{y+1} &= \delta_M(M_y, xacc_t'[y]) \end{aligned}$$

Using notation for memory updates with access sequences, for $y \leq na(t)$ we obtain

$$M_y = \Delta_M^y(M_0, xacc_t'[0 : y-1]).$$

We show a technical result first. Below in this chapter by acc_t' we denote the part of sequence acc' consisting of accesses ending in cycle t, i.e.,

$$acc_t'[0 : ne(t)-1] = acc'[NE(t) : NE(t+1)-1].$$

Lemma 90. *For* $y \in [0 : na(t) - 1]$ *we claim*

$$M_y = \Delta_M^{x(y,t)}(M_0, acc'_t).$$

Proof of lemma 90. The proof is by an easy induction on $y < na(t)$, and therefore we omit it. Note, for sequence of processor accesses $xacc'_t$ by definition we clearly have

$$\forall y \in [0 : na(t) - 1] : \; xacc'_t[y] = acc'_t[x(y,t)]. \qquad \square$$

Next we argue that in cycle t execution of accesses only from sequence $xacc'_t$ gives us correct memory abstraction in cycle $t+1$. For that to show we require the results on the sequential consistency from [KMP14] (lemma 51).

Lemma 91.

$$m(h_\pi^{t+1}) = \Delta_M^{na(t)}(m(h_\pi^t), xacc'_t)$$

Proof of lemma 91. We start by showing that

$$\begin{aligned} m(h_\pi^{t+1}) &= \Delta_M^{NE(t+1)}(m(h_\pi^0), acc') \qquad \text{(lemma 51)} \\ &= \Delta_M^{ne(t)}(\Delta_M^{NE(t)}(m(h_\pi^0), acc'[0 : NE(t) - 1]), acc'[NE(t) : NE(t+1) - 1]) \\ &= \Delta_M^{ne(t)}(m(h_\pi^t), acc'[NE(t) : NE(t+1) - 1]) \qquad \text{(lemma 51)} \\ &= \Delta_M^{ne(t)}(m(h_\pi^t), acc'_t) \qquad \text{(definition).} \end{aligned}$$

Therefore, using the definition of M_0 we are left to prove

$$\Delta_M^{ne(t)}(M_0, acc'_t) = \Delta_M^{na(t)}(M_0, xacc'_t)$$

which can easily be shown using lemma 90 and the fact that accesses from $E(t) \setminus A(t)$ do not change the memory abstraction. $\square$

8.4 Liveness

In this section we proceed to show liveness of our pipelined implementation. Note, the latter result is established in the end of Sect. 8.4.2 (lemma 98). In the remained of this section we develop, in Sects. 8.4.3 and 8.4.4, some more formalism in order to show, in Sect. 8.4.5, that cycles in which instructions progress through the pipeline are unique (equation 65).

8.4.1 Lowest Truly Full Stage

Before we begin the proofs, we introduce one more technical definition. We say that pipeline stage $L(t)$ is the *lowest truly full stage*[4] in the given cycle (t) if

$$L(t) = \max \{k \in [0 : 5] \mid rfull_k^t\}.$$

In order to derive properties of L, we require the following result. Namely, we need to show that the nested MMUs are live while utilized by the pipelined processor.

Lemma 92.

$$(L(t) = 0) \wedge mmu_I^t.treq \quad \rightarrow \quad (\; mmu_I^t.busy \rightarrow \; \exists t' > t : /mmu_I^{t'}.busy \;) \qquad (1)$$

$$(L(t) = 4) \wedge mmu_E^t.treq \quad \rightarrow \quad (\; mmu_E^t.busy \rightarrow \; \exists t' > t : /mmu_E^{t'}.busy \;) \qquad (2)$$

[4] Registers of the memory stage were excluded for technical reasons: since instructions are executed in the memory stage, it suffices to consider only the register stages above.

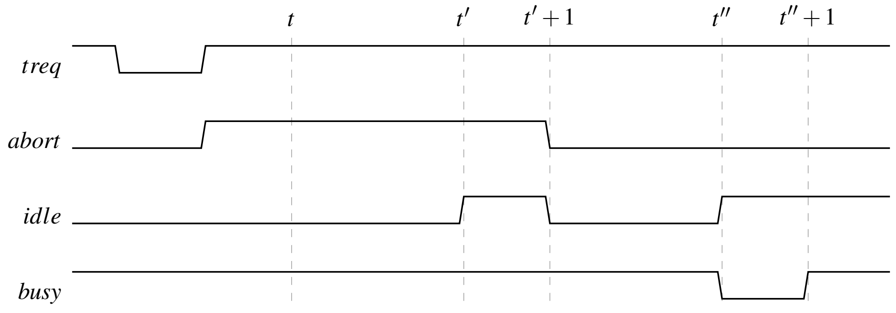


Fig. 38: Timing of the MMU control signals

Proof of lemma 92.1. Timing diagrams from Fig. 38 are meant to illustrate the proof. Directly from lemma 31 we argue for some cycle $t' > t$:

$$idle(mmu_I^{t'}).$$

Clearly, we can assume that mmu_I is still busy in cycle t' (assume that in cycle t mmu_I was busy recovering from an abort, e.g., after a *jisr*), since otherwise the claim follows. From the latter we know the translation request is high in cycle t'. We conclude

$$t\text{-}start(t')$$

and using lemma 29 argue that translation ends in some cycle cycle $t'' > t$.

$$t\text{-}end(t'')$$

Moreover, we argue that the translation is regular

$$treg[t' : t'']$$

since the translation request is never taken away in the latter cycles. Note, referring to the hardware specification of processor from Sect. 8.1.5, for cycles $\tilde{t} \in [t : t'']$ we have:

$$k(\tilde{t}) = L(t) = 0$$

and

$$mmu_I^{\tilde{t}}.treq = /host(\tilde{t}) = mmu_I^t.treq.$$

Since the translation is regular, applying lemma 30 we derive that in cycle t'' there is a hit in the hardware TLB

$$mmu_I^{t''}.tlb.hit$$

which essentially gives the claim. □

Since the proof of the second part is completely analogous, we omit it to shorten the presentation. Next we show that rollback request signals are not generated in stages below $L(t)$.

Lemma 93.

$$R(t) \leq L(t)$$

Proof of lemma 93. By contradiction. Assume that the rollback request signal in stage $L(t)+1$ is active in cycle t. We derive

$$\begin{aligned} rbr^t_{L(t)+1} &\rightarrow \begin{cases} misspec^t_2 \vee misspec^t_5 & L(t) < 2 \\ misspec^t_5 & 2 \leq L(t) < 5 \end{cases} \\ &\rightarrow \begin{cases} rfull^t_2 \vee rfull^t_5 & L(t) < 2 \\ rfull^t_5 & L(t) < 5 \end{cases} \\ &\rightarrow \begin{cases} L(t) \geq 2 & L(t) < 2 \\ L(t) = 5 & L(t) < 5 \end{cases} \end{aligned}$$

which is a contradiction. □

In the following lemma we argue there is a cycle $t' \geq t$ in which the stage immediately below $L(t)$ is updated.

Lemma 94.

$$/ue^t_{L(t)+1} \rightarrow \exists t' > t : ue^{t'}_{L(t)+1}$$

Proof of lemma 94. Directly from the definitions we derive

$$\begin{aligned} ue^t_{L(t)+1} &= rfull^t_{L(t)} \wedge /stall^t_{L(t)+1} \wedge /rbr^t_{L(t)+1} \\ &= /stall^t_{L(t)+2} \wedge /haz^t_{L(t)+1} \end{aligned} \tag{45}$$

since there are no active rollback request signals in stages below $L(t)$ (lemma 93). Moreover, for the stall signals in stages below $L(t)+1$ we conclude

$$\begin{aligned} stall^t_{L(t)+2} &\rightarrow full^t_{L(t)+1} \\ &\rightarrow rbp^t_{L(t)+1} \end{aligned} \tag{46}$$

since otherwise we obtain a contradiction.

$$full^t_{L(t)+1} \wedge /rbp^t_{L(t)+1} \rightarrow rfull^t_{L(t)+1}$$

Referring to Sect. 8.1.5, where the processor components were interconnected, we argue by case split on the lowest truly full stage:

- $L(t) = 0$. First, assume that the stall signal of stage IT is low in cycle t.

  $$stall^t_2 = 0$$

  For the instruction address translation stage we have:

  $$\begin{aligned} haz_1 &= treq_I \wedge (mmu_I.busy \vee inval(mmu_I)) \vee drain \\ &= mmu_I.treq \wedge mmu_I.busy. \end{aligned}$$

  Therefore, in case mmu_I is not requested or not busy in cycle t, there is nothing to show. Otherwise, we argue using liveness of the nested MMU (lemma 92) and conclude the claim for some cycle $t' > t$.

  In the presence of the stall signal of stage IT in cycle t we split cases on whether the rollback hazard of stage IT is active in cycle t.

  – $rhaz^t_2 = 0$. In this case, from the construction of the stall engine we derive

  $$\begin{aligned} /rhaz^t_2 &\rightarrow /rbp^{t+1}_1 \\ &\rightarrow /stall^{t+1}_2 \qquad \text{(equation 46)} \end{aligned}$$

  and the claim follows exactly as in the case above for some cycle $t' \geq t+1$.

  – $rhaz^t_2 = 1$. Otherwise, in case the instruction cache is busy in cycle t, from liveness of the cache memory system [KMP14] for some cycle $t' > t$ we conclude

  $$rhaz^{t'}_2 = 0$$

  and the claim follows as above, for some cycle $t'' \geq t'+1$.

Using specifics of the design of our pipeline, we derive:

$$\begin{aligned} stall^t_{L(t)+2} &\rightarrow rbp^t_{L(t)+1} \qquad \text{(equation 46)} \\ &\rightarrow L(t)+1 = 1 \qquad \text{(invariant 9)} \\ &\rightarrow L(t) = 0. \end{aligned}$$

Therefore, according to equation 45, for stages $L(t) > 0$ we simply have

$$ue^t_{L(t)+1} = /haz^t_{L(t)+1}.$$

- $L(t) = 1$. For the instruction fetch stage we argue solely from liveness of the cache memory system. In case the instruction cache is not busy, there is nothing to show. Otherwise, we conclude the claim for some cycle $t' > t$. Note, the operating conditions of the instruction cache are always respected by the new stall engine (see Sect. 8.1.1).
- $L(t) = 2$. For the instruction decode stage there is nothing to show, since there cannot be any forwarding hits in stages below $L(t)$.

$$\begin{aligned} haz_3^t &\leftrightarrow haz_A^t \vee haz_B^t \\ &\rightarrow \exists k \in [3:5] : \ hit_A^t[k] \vee hit_B^t[k] \\ &\rightarrow L(t) \geq 3 \end{aligned}$$

- $L(t) = 3$. For the execution stage there is nothing to show since the stage never produces hazard signals.
- $L(t) = 4$. For the effective address translation stage we argue exactly as above, for the instruction address translation. Namely, we show

$$haz_5 = mmu_E.treq \wedge mmu_E.busy$$

  and either there is nothing to show (in case mmu_E is not requested or not busy in cycle t), or we argue using lemma 92 to conclude the claim for some cycle $t' > t$.
- $L(t) = 5$. For the memory stage we argue as for the instruction fetch above. Using liveness of the cache memory system we conclude the claim for some cycle $t' > t$ in case the data cache is requested and busy in cycle t. Otherwise, there is nothing to show. □

8.4.2 Liveness of Pipeline Stages

In the next lemma we show that the lowest truly full stage "creeps down" in cycles t in which the stage immediately below $L(t)$ is updated.

Lemma 95. *Assume* $L(t) < 5$.

$$L(t+1) = L(t) + ue_{L(t)+1}^t$$

Proof of lemma 95. From the definition of the lowest truly full stage for cycle t we know that i) stages below $L(t)$ are not truly full and ii) stages below $L(t)+1$ are not updated.

$$\forall k > L(t) : /rfull_k^t \wedge /ue_{k+1}^t \tag{47}$$

We split cases on whether the stage immediately below $L(t)$ is updated in cycle t:

- $ue_{L(t)+1}^t = 1$. From above (equation 47) and construction of the stall engine for stages below $L(t)+1$ we clearly have

$$\forall k > L(t)+1 : /rfull_k^t.$$

  By definition we therefore have

$$L(t+1) \leq L(t)+1.$$

  Construction of the control mechanisms (lemma 75) ensures that stage $L(t)+1$ is truly full in cycle $t+1$ ($rfull_{L(t)+1}^{t+1}$). The latter obviously gives the claim, since

$$L(t+1) \geq L(t)+1.$$

- $ue_{L(t)+1}^t = 0$. Together with equation 47, from the construction of the stall engine for stages below $L(t)$ we clearly have

$$\forall k > L(t) : /rfull_k^t,$$

  which by definition gives

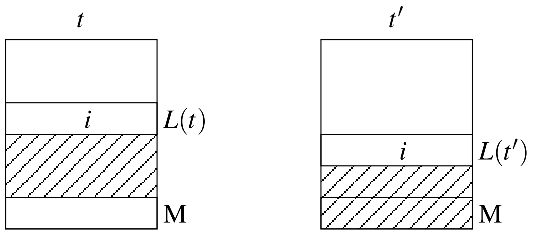


Fig. 39: Empty pipeline stages (shaded) below the lowest truly full stages in cycles t and $t' > t$. Note, instruction $i = I(L(t),t)$ advances further down through the pipeline and in cycle t' is found in stage $L(t')$.

$$L(t+1) \leq L(t).$$

From lemma 93 we know

$$rbr^{t}_{L(t)+1} = 0.$$

Again from the construction of the stall engine (part (5) of lemma 75): the real full bit is not cleared in the absence of rollback requests from below.

$$rfull^{t}_{L(t)} \wedge /ue^{t}_{L(t)+1} \wedge /rbr^{t}_{L(t)+1} \rightarrow rfull^{t+1}_{L(t)}$$

The claim obviously follows, since

$$L(t+1) \geq L(t). \qquad \square$$

Next we show that the lowest truly full stage "creeps down" over time *unless* it is already "at the bottom". Figure 39 is included in order to illustrate the proof.

Lemma 96. *Assume $L(t) < 5$.*

$$\exists t' > t : \; L(t') = L(t) + 1$$

Proof of lemma 96. From lemma 94 we know the stage immediately below $L(t)$ is updated in some cycle $t' \geq t$.

$$ue^{t'}_{L(t)+1}$$

Moreover, for the first such cycle we know that stage $L(t)+1$ is not updated in cycles $\tilde{t} \in [t : t'-1]$.

$$/ue^{\tilde{t}}_{L(t)+1}$$

Applying lemma 95 we obtain

$$L(t') = L(t)$$

and

$$L(t'+1) = L(t') + 1 = L(t) + 1. \qquad \square$$

From the latter lemma one can easily derive the following property.

$$k \in [L(t)+1 : 5] \rightarrow \exists t' > t : \; L(t') = k \tag{48}$$

In the following lemma we state that the memory stage is updated infinitely often.

Lemma 97.

$$/ue^{t}_{6} \rightarrow \exists t' > t : \; ue^{t'}_{6}$$

Proof of lemma 97. By case split on the lowest truly full stage:

- $L(t) = 5$. Since by the assumption the memory stage is not updated in cycle t, the claim follows directly from lemma 94. For some cycle $t' > t$ we conclude

$$ue^{t'}_{5+1}.$$

- $L(t) < 5$. Using equation 48 we obtain for some cycle $t' > t$

$$L(t') = 5.$$

The claim follows as above, from lemma 94. For some cycle $t'' \geq t'$ we conclude

$$ue^{t''}_{5+1}. \qquad \square$$

Recall, since the memory stage is never rolled-back ($/rbr_6$) in our implementation, the following holds.

$$I(6,t+1) = I(6,t) + ue^t_6 \tag{49}$$

Finally, we show that the pipelined implementation is live, i.e., we show that every ISA instruction is eventually executed.

Lemma 98.

$$\forall i\ \exists t:\ i = I(6,t) \wedge ue^t_6$$

Proof of lemma 98. By induction on index i of the instruction in the memory stage. For the induction base ($i = 0$) recall that immediately after reset we have by definition

$$I(6,0) = 0.$$

From lemma 97 we know the memory stage is updated infinitely often. Consider the first such cycle $t' > 0$ in which the memory stage is updated. Clearly, we have

$$(\forall t \in [0 : t'-1] : /ue^t_6) \wedge ue^{t'}_6.$$

Using equation 49 for the scheduling function we conclude:

$$I(6,t') = I(6,0) = 0.$$

For the induction step from i to $i+1$ we argue as follows. Directly from the induction hypothesis we know there is a cycle t such that

$$i = I(6,t) \wedge ue^t_6.$$

From the definition of the scheduling functions we immediately obtain

$$I(6,t+1) = i+1.$$

Applying lemma 97 we know there is a cycle $t' \geq t+1$:

$$ue^t_6 \wedge (\forall \tilde{t} \in [t+1 : t'-1] : /ue^{\tilde{t}}_6) \wedge ue^{t'}_6.$$

To complete the induction step we argue as above, using equation 49:

$$I(6,t') = I(6,t+1) = i+1. \qquad \square$$

In Sect. 8.4.5 we are to show uniqueness of cycles in which instructions in circuit stages below μ progress down the pipeline. The remainder of this section is spent to derive some of the necessary arguments. Thus, below we show that every pipeline stage is updated infinitely often.

Lemma 99.

$$/ue^t_k \ \rightarrow\ \exists t' > t:\ ue^t_k$$

Proof of lemma 99. For the write back stage ($k = 7$) we assume $/ue_7^t$, since otherwise there is nothing to show. From the construction of the stall engine we derive

$$rfull_6^t = 0.$$

From lemma 97 we know that the memory stage is updated in cycle $t' > t$ ($ue_6^{t'}$), and therefore

$$rfull_6^{t'+1} = 1$$

and the claim follows for $t'' = t' + 1$.
For the remaining stages by induction on $\ell \leq 5$ we proceed to show

$$/ue_{6-\ell}^t \ \rightarrow\ \exists t' > t:\ ue_{6-\ell}^{t'}.$$

The base case ($\ell = 0$) is exactly the statement of lemma 97. For the induction step from ℓ to $\ell + 1 \leq 5$ we argue by contradiction as follows. From the induction hypothesis for some cycle $t' \geq t$ we have

$$ue_{6-\ell}^{t'} = 1.$$

Further we assume

$$ue_{5-\ell}^{t'} = 0,$$

since otherwise there is nothing to show, and from the construction of the stall engine (part (6) of lemma 75) we conclude

$$rfull_{5-\ell}^{t'+1} = 0. \tag{50}$$

Again, from the induction hypothesis for some cycle $t'' > t' + 1$ we have

$$ue_{6-\ell}^{t''} = 1$$

which by definition of the update enable signal implies

$$rfull_{5-\ell}^{t''} = 1. \tag{51}$$

By contradiction we argue

$$\exists \tilde{t} \in [t'+1 : t''-1]:\ ue_{5-\ell}^{\tilde{t}}.$$

Thus, we proceed to show

$$\forall \tilde{t} \in [t'+1 : t'']: /rfull_{5-\ell}^{\tilde{t}}$$

by induction on θ, where θ denotes the length of sub-interval

$$[t'+1 : t'+\theta] \subseteq [t'+1 : t''].$$

For the base case ($\theta = 0$) there is nothing to show (equation 50). For the induction step from θ to $\theta + 1 \leq t'' - t'$ we argue as follows. For $\bar{t} = t' + \theta$, using part (4) of lemma 75 we derive

$$/rfull_{5-\ell}^{\bar{t}} \wedge /ue_{5-\ell}^{\bar{t}} \ \rightarrow\ /rfull_{5-\ell}^{\bar{t}+1}$$

which completes the proof; the contradiction immediately follows (equation 51). □

The following existence result is necessary already in the next section (Sect. 8.4.3).

$$\forall i\, \forall t:\ (\forall k:\ I(k+1,t) \neq i) \vee (\exists!\, k:\ I(k+1,t) = i \wedge rfull_k^t) \tag{52}$$

Proof of equation 52. Equivalently, we proceed to show by contradiction the following statement.

$$\exists k:\ I(k+1,t) = i \ \rightarrow\ \exists!\, \tilde{k}:\ I(\tilde{k}+1,t) = i \wedge rfull_{\tilde{k}}^t$$

Assume that in cycle t stages k and $\tilde{k} > k$ are truly full

$$rfull_k^t \wedge rfull_{\tilde{k}}^t$$

with instruction

$$I(k+1,t) = I(\tilde{k}+1,t).$$

The contradiction follows.

$$\begin{aligned} I(k+1,t) &= I(\tilde{k}+1,t) + \textstyle\sum_{j=k+1}^{\tilde{k}} rfull_j^t \qquad \text{(lemma 81)} \\ &\geq I(\tilde{k}+1,t) + rfull_{\tilde{k}}^t \\ &> I(\tilde{k}+1,t) \end{aligned}$$

□

8.4.3 Instruction Stage

Truly full pipeline stages containing in cycle t (in the circuit stages below) instruction i are called *instruction stages* of instruction i in the given cycle (t):

$$P(i,t) = \{k \mid I(k+1,t) = i \wedge rfull_k^t\}.$$

The latter stages (if they exist) are provably unique (see Sect. 8.4.2, equation 52).

$$\pi(i,t) = \begin{cases} \varepsilon\, P(i,t) & P(i,t) \neq \emptyset \\ +\infty & \text{otherwise} \end{cases}$$

Note, if instruction i is not contained anywhere in the pipeline in cycle t, stage $\pi(i,t)$ is chosen to be $+\infty$ merely for convenience. In the following lemma we derive the value of stage π for a given instruction in the next cycle.

Lemma 100. *Assume $\pi(i,t) \leq 5$ and $R(t) \leq \pi(i,t)$.*

$$\pi(i,t+1) = \pi(i,t) + ue^t_{\pi(i,t)+1}$$

Proof of lemma 100. From the definition of the instruction stage $\pi = \pi(i,t)$ we have

$$I(\pi+1,t) = i \wedge rfull_\pi^t.$$

Next we split cases on whether the stage below π is updated in cycle t or not:

- $ue^t_{\pi+1} = 1$. In this case we proceed to show that the instruction stage advances.

  $$I(\pi+2,t+1) = i \wedge rfull^{t+1}_{\pi+1}$$

  For the scheduling function in stage $\pi+2$ we derive

  $$\begin{aligned} I(\pi+2,t+1) &= \begin{cases} I(\pi+1,t) & rfull^t_{\pi+1} \\ I(\pi+2,t) & \text{otherwise} \end{cases} && \text{(lemma 75.3; definition)} \\ &= I(\pi+1,t) && \text{(lemma 79).} \end{aligned}$$

  For the real full bit of stage $\pi+1$ the claim follows from part (1) of lemma 75.
- $ue^t_{\pi+1} = 0$. In this case we are to show that the instruction stage does not change.

  $$I(\pi+1,t+1) = i \wedge rfull^{t+1}_\pi$$

  For the scheduling function in stage $\pi+1$ by definition we have

  $$I(\pi+1,t+1) = I(\pi+1,t)$$

  and for the real full bit of stage π we argue using part (5) of lemma 75.

  $$rfull^t_\pi \wedge /ue^t_{\pi+1} \wedge /rbr^t_{\pi+1} \;\rightarrow\; rfull^{t+1}_\pi \qquad \square$$

In the following lemma we show that rollback request signals are not generated in stages below $\mu(t)$.

Lemma 101.

$$R(t) \leq \mu(t)$$

Proof of lemma 101. Assume

$$R(t) > 0$$

since otherwise there is nothing to show. From the definition we know

$$R(t) = k \in \{2,5\}.$$

Instruction in circuit stage k is clearly not live in cycle t

$$rbr^t_k \;\rightarrow\; /live(k,t)$$

which by definition implies

$$\mu(t) \geq k. \qquad \square$$

In order to show later that μ takes values only in range $[0:5]$, we require the following auxiliary result.

Lemma 102. *Assume* $\mu(t) = 5$.

$$ue_6^t \rightarrow rbr_5^t$$

Proof of lemma 102. By contradiction; we assume

$$\mu(t) = 5 \wedge ue_6^t \wedge /rbr_5^t.$$

From the scheduling functions by definition we have

$$I(6,t+1) = I(5,t).$$

The latter by lemmas 98 and 82 for some $t' \geq t+1$ gives

$$I(6,t') = I(6,t+1) \wedge ue_6^{t'}$$

which by definition implies

$$live(6,t+1).$$

Repeating the arguments presented in the proof of lemma 86 we derive

$$live(5,t)$$

which is a contradiction. □

In case not all stages are live in cycle t, the value of μ in cycle $t+1$ is given in the following lemma.

Lemma 103. *Assume* $\mu(t) > 0$.

$$\overline{ue_{\mu(t)+1}^t} \rightarrow \mu(t+1) = \mu(t) \tag{1}$$

$$ue_{\mu(t)+1}^t \rightarrow \mu(t+1) = \begin{cases} 0 & rbr_{\mu(t)}^t \\ \mu(t)+1 & otherwise \end{cases} \tag{2}$$

Proof of lemma 103.1. From lemmas 84 and 101 we resp. know

$$rfull_{\mu(t)}^t = 1$$

and

$$rbr_{\mu(t)+1}^t = 0.$$

Since the stage below $\mu(t)$ is neither updated nor rolled-back in cycle t, by definition of the scheduling functions we have

$$I(\mu(t)+1,t+1) = I(\mu(t)+1,t).$$

From the definition of μ we clearly have

$$live(\mu(t)+1,t)$$

and therefore

$$live(\mu(t)+1,t+1).$$

From the arguments above we conclude

$$\mu(t+1) < \mu(t)+1.$$

From the construction of the stall engine (part (3) of lemma 75) we know that stage $\mu(t)$ is not updated in cycle t.

$$ue_{\mu(t)}^t = 0$$

Next we split cases on the value of the rollback request signal in stage $\mu(t)$ in cycle t:

- $rbr^t_{\mu(t)} = 0$. In this case, from the definition of the scheduling functions we have
$$I(\mu(t), t+1) = I(\mu(t), t).$$
By definition stage $\mu(t)$ is not live in cycle t
$$/live(\mu(t), t)$$
which implies
$$/live(\mu(t), t+1).$$
From the arguments above we conclude
$$\mu(t+1) \geq \mu(t).$$
- $rbr^t_{\mu(t)} = 1$. In this case we proceed to show that the rollback request signal of stage $\mu(t)$ remains active in cycle $t+1$.
$$rbr^t_{\mu(t)} \wedge /ue^t_{\mu(t)+1} \rightarrow rbr^{t+1}_{\mu(t)} \tag{53}$$
Proof of equation 53. From the definition of the update enable signal we know that stage $\mu(t)$ is not overwritten in cycle t.
$$ue^t_{\mu(t)} = 0 \tag{54}$$
Using lemma 101 we derive
$$\mu(t) = R(t) \in \{2, 5\}.$$
In case $\mu(t) = 2$, we argue using the definition of signal $misspec_2$ as follows.
$$\begin{aligned} rbr^t_2 &\rightarrow rfull^t_2 \wedge iret^t_2 && \text{(interconnect)} \\ &\rightarrow rfull^{t+1}_2 \wedge iret^{t+1}_2 && \text{(lemma 75.5; equation 54)} \\ &\rightarrow rbr^{t+1}_2 && \text{(interconnect)} \end{aligned}$$
In the other case, if $\mu(t) = 5$, we derive a contradiction.
$$\begin{aligned} rbr^t_5 &\rightarrow jisr(t) && \text{(interconnect)} \\ &\rightarrow ue^t_6 && \text{(definition)} \end{aligned}$$
□

Therefore, we conclude
$$\begin{aligned} \mu(t+1) &\geq R(t+1) && \text{(lemma 101)} \\ &\geq \mu(t) && \text{(equation 53)} \end{aligned}$$
and the claim follows. □

In the proof lines below we find the following abbreviations useful.
$$\begin{aligned} empty{\uparrow}(k,t) &\leftrightarrow \forall j \in [1:k] : /rfull^t_j \\ empty{\downarrow}(k,t) &\leftrightarrow \forall j \in [k:5] : /rfull^t_j \end{aligned}$$

Proof of lemma 103.2. First consider the case with an active rollback request signal in stage $\mu(t)$ in cycle t ($rbr^t_{\mu(t)}$). Using lemma 101 we derive
$$\mu(t) = R(t) \in \{2, 5\}$$
and split cases on the value of μ in cycle t:

- $\mu(t) = 5$. For the instruction in stage 1 in cycle $t+1$, from lemmas 98 and 82 we obtain

$$\exists t' \geq t+1:\ I(6,t') = I(1,t+1) \wedge ue_6^{t'}. \tag{55}$$

Also for cycles in which instruction $i = I(1,t+1)$ progresses down through the pipeline we argue that the following holds.

$$\forall \tilde{t} \in [t+1 : t']:\ empty{\downarrow}(\pi(i,\tilde{t})+1,\tilde{t}) \tag{56}$$

Proof of equation 56. By induction on θ, where θ is the "length" of sub-interval

$$[t+1 : t+1+\theta] \subseteq [t+1 : t'].$$

For the base case ($\theta = 0$) the claim follows from the arguments above: after the *jisr* in cycle t we have

$$\pi(i,t+1) = 0$$

and by lemma 76 we get

$$empty{\downarrow}(1,t+1).$$

For the induction step from θ to $\theta+1 < t'-t$ we abbreviate

$$\tilde{t} = t+1+\theta$$

and split cases on whether the stage below $\pi(i,\tilde{t})$ is updated in cycle $\tilde{t}$ or not. In case

$$ue^{\tilde{t}}_{\pi(i,\tilde{t})+1} = 1$$

from lemma 100 we have

$$\pi(i,\tilde{t}+1) = \pi(i,\tilde{t})+1.$$

From the induction hypothesis and definition of the update enable signal, for

$$\pi \in [\pi(i,\tilde{t})+2 : 5]$$

by part (4) of lemma 75 we argue

$$/rfull^{\tilde{t}}_{\pi} \wedge /ue^{\tilde{t}}_{\pi} \ \rightarrow\ /rfull^{\tilde{t}+1}_{\pi}$$

which by definition gives the claim.

$$empty{\downarrow}(\pi(i,\tilde{t}+1)+1,\tilde{t}+1)$$

Otherwise, if the stage below $\pi(i,\tilde{t})$ is not updated in cycle $\tilde{t}$

$$ue^{\tilde{t}}_{\pi(i,\tilde{t})+1} = 0$$

we proceed to show the following auxiliary result.

$$empty{\downarrow}(\pi(i,\tilde{t})+1),\tilde{t}) \ \rightarrow\ R(\tilde{t}) \leq \pi(i,\tilde{t}) \tag{57}$$

(Note, the proof below is illustrated in Fig. 40(a).)

Proof of equation 57. By contradiction. Assume that the rollback request signal is active in cycle $\tilde{t}$ at stage

$$r = R(\tilde{t}) > \pi(i,\tilde{t}).$$

From the hardware interconnect (Sect. 8.1.5, definition of the *misspec* signals) we know that stage $r \leq 5$ has a real full bit in cycle $\tilde{t}$.

$$rfull^{\tilde{t}}_{r} = 1$$

The latter contradicts the assumption that there are no truly full stages between stages $\pi(i,\tilde{t})$ and the memory stage in cycle $\tilde{t}$.

$$\forall j \in [\pi+1 : 5] : /rfull^{\tilde{t}}_{j} \qquad \square$$

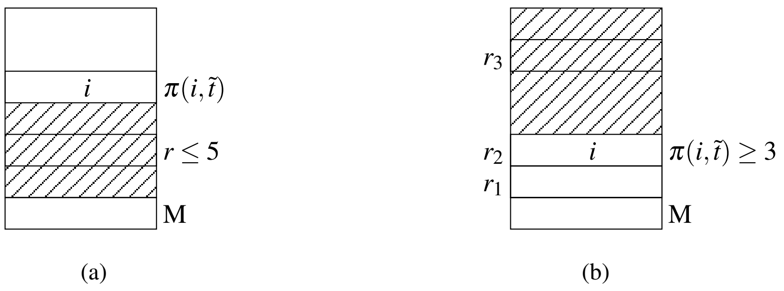


Fig. 40: Possible rollback request signals in cycle $\tilde{t}$

Using the result above and lemma 100 we get

$$\pi(i,\tilde{t}+1) = \pi(i,\tilde{t}).$$

Repeating the arguments from the case above ($ue^{\tilde{t}}_{\pi(i,\tilde{t})+1}$), for stages

$$\pi \in [\pi(i,\tilde{t})+1 : 5]$$

we analogously derive

$$rfull^{\tilde{t}+1}_{\pi} = 0$$

which completes the induction step, and therefore the proof. □

From equations 56 and 57 we easily conclude the absence of rollback request signals for instruction $i = I(1,t+1)$ while it progresses down through the pipeline.

$$\forall \tilde{t} \in [t+1 : t'] : /rbr^{\tilde{t}}_{\pi(i,\tilde{t})+1} \tag{58}$$

From the result above we clearly have

$$\forall \tilde{t} \in [t+1 : t'] : \; I(k,\tilde{t}) = I(1,t+1) \rightarrow /rbr^{\tilde{t}}_{k}.$$

Combining the statement above with equation 55 we conclude

$$live(1,t+1)$$

and the claim follows by lemma 83.

- $\mu(t) = 2$. Since stage 3 is updated in cycle t, from lemma 85 we obtain

$$live(4,t+1)$$

which by definition is

$$\begin{aligned} &\exists t' \geq t+1 : \; I(6,t') = I(4,t+1) \wedge ue^{t'}_{6} \; \wedge \\ &\forall \tilde{t} \in [t+1 : t'] \; \forall \tilde{k} \in [4:6] : \; I(\tilde{k},\tilde{t}) = I(4,t+1) \rightarrow /rbr^{\tilde{t}}_{\tilde{k}}. \end{aligned}$$

Moreover, using the definition of signal $misspec_2$ we argue that in cycle $t+1$ stage 3 has a real full bit and contains the data of a legal *eret* instruction:

$$\begin{aligned} rbr^{t}_{2} &\rightarrow rfull^{t}_{2} \wedge iret^{t}_{2} && \text{(interconnect)} \\ &\rightarrow rfull^{t+1}_{3} \wedge iret^{t+1}_{3} && \text{(lemma 75.1).} \end{aligned}$$

For cycles in which instruction $i = I(4,t+1)$ progresses down through the pipeline we proceed to show the following.

$$\forall \tilde{t} \in [t+1 : t'+1] : \; live(\pi(i,\tilde{t})+1,\tilde{t}) \wedge \pi(i,\tilde{t}) \geq 3 \wedge iret^{\tilde{t}}_{\pi(i,\tilde{t})} \tag{59}$$

Proof of equation 59. By induction on θ, where θ is the “length” of sub-interval

$$[t+1:t+1+\theta] \subseteq [t+1:t'+1].$$

For the base case ($\theta = 0$) the claim follows from the arguments above.

$$live(4,t+1) \wedge \pi(i,t+1) = 3 \wedge iret_3^{t+1}$$

For the induction step from θ to $\theta+1 \leq t'-t$ we abbreviate

$$\tilde{t} = t+1+\theta$$

and split cases on whether the stage below $\pi(i,\tilde{t})$ is updated in cycle $\tilde{t}$ or not. In case

$$ue^{\tilde{t}}_{\pi(i,\tilde{t})+1} = 1$$

from lemma 100 we have

$$\pi(i,\tilde{t}+1) = \pi(i,\tilde{t})+1$$

and using the induction hypothesis we conclude

$$\pi(i,\tilde{t}+1) \geq 3 \wedge iret^{\tilde{t}+1}_{\pi(i,\tilde{t}+1)}.$$

For the scheduling function in the stage below $\pi = \pi(i,\tilde{t}+1)$ we derive

$$\begin{aligned} I(\pi+1,\tilde{t}+1) &= \begin{cases} I(\pi,\tilde{t}) & rfull^{\tilde{t}}_{\pi} \\ I(\pi+1,\tilde{t}) & \text{otherwise} \end{cases} && \text{(lemma 75.3; definition)} \\ &= I(\pi+1,\tilde{t}) && \text{(lemma 79).} \end{aligned}$$

From the induction hypothesis and lemma 83 we conclude

$$live(\pi+1,\tilde{t})$$

which implies

$$live(\pi+1,\tilde{t}+1).$$

Otherwise, if the stage below $\pi(i,\tilde{t})$ is not updated in cycle $\tilde{t}$

$$ue^{\tilde{t}}_{\pi(i,\tilde{t})+1} = 0$$

we proceed to show the following auxiliary result.

$$live(\pi(i,\tilde{t})+1,\tilde{t}) \wedge \pi(i,\tilde{t}) \geq 3 \wedge iret^{\tilde{t}}_{\pi(i,\tilde{t})} \rightarrow R(\tilde{t}) = 0 \tag{60}$$

(Note, the proof below is illustrated in Fig. 40(b).)

Proof of equation 60. By contradiction. Assume that the rollback request signal is active in cycle $\tilde{t}$ at stage

$$r = R(\tilde{t}) > 0.$$

Next we split cases on the value of $r \leq 5$:

- $r = r_1 > \pi(i,\tilde{t})$. From the definition of the rollback request signals we conclude

$$rbr^{\tilde{t}}_{\pi(i,\tilde{t})+1} = 1$$

  which contradicts out first assumption (see Sect. 8.2.3):

$$rbr^{\tilde{t}}_{\pi(i,\tilde{t})+1} \rightarrow /live(\pi(i,\tilde{t})+1,\tilde{t}).$$

- $r = r_2 = \pi(i,\tilde{t})$. From the definition of the rollback request signals we conclude

$$misspec_r^{\tilde{t}} = 1.$$

From the second and the third assumption we have

$$r \geq 3 \wedge rfull_r^{\tilde{t}} \wedge iret_r^{\tilde{t}}$$

which gives a contradiction (see Sect. 8.1.5, definition of the *misspec* signals).

$$misspec_r^{\tilde{t}} = 0$$

- $r = r_3 < \pi(i,\tilde{t})$. Again, from the definition of *misspec* signals we know that stage r has a real full bit in cycle $\tilde{t}$.

$$rfull_r^{\tilde{t}} = 1$$

From the second and the third assumption for $k = \pi(i,\tilde{t})$ we have

$$k \geq 3 \wedge rfull_k^{\tilde{t}} \wedge iret_k^{\tilde{t}}$$

and using part (1) of lemma 77 we derive a contradiction:

$$\forall j \in [1 : k-1] : /rfull_j^{\tilde{t}}. \qquad \square$$

Using the result above and lemma 100 we get

$$\pi(i,\tilde{t}+1) = \pi(i,\tilde{t})$$

which by the induction hypothesis implies

$$\pi(i,\tilde{t}+1) \geq 3 \wedge iret_{\pi(i,\tilde{t}+1)}^{\tilde{t}+1}.$$

For the scheduling function in the stage below $\pi = \pi(i,\tilde{t})$ by definition we have

$$I(\pi+1,\tilde{t}+1) = I(\pi+1,\tilde{t}).$$

Finally, from the induction hypothesis we have

$$live(\pi+1,\tilde{t})$$

which implies

$$live(\pi+1,\tilde{t}+1). \qquad \square$$

From equations 59 and 60 we easily conclude the absence of any rollback request signals while instruction $i = I(4,t+1)$ progresses down through the pipeline.

$$\forall \tilde{t} \in [t+1 : t'+1] : /rbr_1^{\tilde{t}} \tag{61}$$

For the instruction in stage 1 in cycle $t+1$, from lemmas 98 and 82 we obtain

$$\exists t'' \geq t+1 : \ I(6,t'') = I(1,t+1) \wedge ue_6^{t''}. \tag{62}$$

Moreover, in cycles in which instruction $I(1,t+1)$ progresses down through the pipeline, repeating the arguments analogous to those presented in the case above (equations 56–58) we conclude

$$\forall \tilde{t} \in [t'+2 : t''] : \ I(k,\tilde{t}) = I(1,t+1) \ \rightarrow \ /rbr_k^{\tilde{t}}$$

which together with equation 61 gives

$$\forall \tilde{t} \in [t+1 : t''] : \ I(k,\tilde{t}) = I(1,t+1) \rightarrow /rbr_k^{\tilde{t}}.$$

Combining the statement above with equation 62 we conclude

$$live(1,t+1)$$

and the claim follows by lemma 83.

Finally we cover the case in which the rollback request signal in stage $\mu(t)$ is inactive in cycle t ($/rbr^t_{\mu(t)}$). Since stage $\mu(t)+1$ is updated in cycle t, from lemma 85 we obtain

$$live(\mu(t)+2, t+1)$$

which clearly gives

$$\mu(t+1) < \mu(t)+2.$$

Moreover, from lemma 102 we know

$$\mu(t) < 5$$

which by lemma 86 gives

$$/live(\mu(t)+1, t+1).$$

From the arguments above we have

$$\mu(t+1) \geq \mu(t)+1$$

and the claim follows. □

For convenience, we generalize the last result as follows.

Lemma 104.

$$\mu(t+1) \leq \mu(t) + ue^t_{\mu(t)+1}$$

Proof of lemma 104. For $\mu(t) > 0$ the claim follows directly from lemma 103. For $\mu(t) = 0$ we proceed to show

$$\mu(t+1) \leq ue^t_1.$$

In case stage 1 is updated in cycle t (ue^t_1), from lemma 85 we obtain

$$live(2, t+1)$$

which by lemma 83 gives the claim.

$$\mu(t+1) < 2$$

Otherwise, if stage 1 is not updated in cycle t ($/ue^t_1$), from lemma 101 we derive

$$R(t) = 0$$

and thus stage 1 is also not rolled-back in cycle t ($/rbr^t_1$). Therefore, from the definition of the scheduling functions we have

$$I(1, t+1) = I(1, t).$$

From the definition of μ we clearly have

$$live(1, t)$$

which implies

$$live(1, t+1).$$

The claim follows by lemma 83.

$$\mu(t+1) < 1$$

□

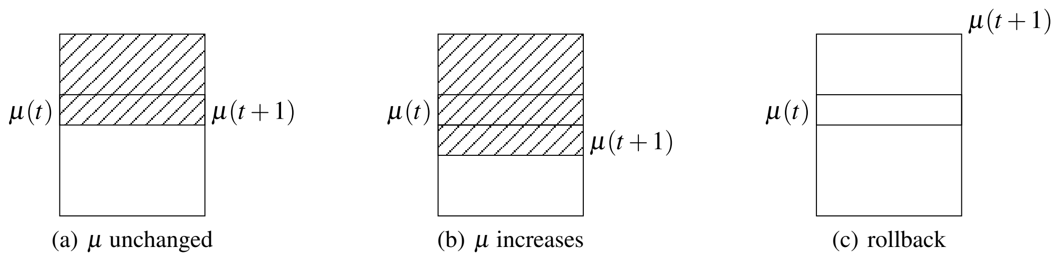


Fig. 41: Pipeline stages target for the induction step in each of the three cases considered in parts (1) (not shaded) and (2) (shaded)

8.4.4 Actual Instruction Index

Due to possible misspeculations, which lead to rollbacks of the affected stages, scheduling functions can be reset to potentially lower values. In order to show that live instructions leave all pipeline stages only once, we introduce an alternative way to index the instructions in the pipeline. The *actual instruction index* below counts updates only of the live stages.

$$\begin{aligned} i(k,0) &= 0 \\ i(k,t+1) &= \begin{cases} i(k,t)+1 & ue_k^t \wedge k > \mu(t) \\ i(k,t) & \text{otherwise} \end{cases} \end{aligned}$$

In the following lemma we relate the scheduling functions with the actual instruction index of the corresponding pipeline stages.

Lemma 105.

$$k > \mu(t) \;\rightarrow\; i(k,t) = I(k,t) \tag{1}$$
$$k \le \mu(t) \;\rightarrow\; i(k,t) = I(\mu(t),t) \tag{2}$$

(Note, the proof below is illustrated in Fig. 41.)

Proof of lemma 105. By induction on the number of hardware cycles t. For the base case ($t=0$) we argue that

$$\mu(0) = 0$$

and there is nothing to show for part (2). For part (1), for all stages k we have

$$i(k,0) = 0 = I(k,0).$$

For the induction step from t to $t+1$ we argue as follows. For part (1), according to lemmas 103 and 104 there are three cases to cover:

- $\mu(t+1) = \mu(t)$. For stages $k > \mu(t)$, using lemma 101 we have

  $$k > R(t)$$

  and argue as follows.

  $$\begin{aligned} i(k,t+1) &= i(k,t) + ue_k^t && \text{(definition)} \\ &= I(k,t) + ue_k^t && \text{(induction hypothesis)} \\ &= I(k,t+1) && \text{(lemma 80)} \end{aligned}$$

- $\mu(t+1) = \mu(t)+1$. For stages $k > \mu(t)+1$ the argument is exactly as above.

- $\mu(t+1) = 0 \neq \mu(t)$. Again, for stages $k > \mu(t)$ the argument is exactly as above. For stages $k \leq \mu(t)$ we argue as follows. According to lemma 103 we have

$$rbr^t_{\mu(t)}$$

which by lemma 101 implies

$$\mu(t) = R(t).$$

To complete the induction step for part (1) we derive:

$$\begin{aligned} i(k,t+1) &= i(k,t) && \text{(definition)} \\ &= I(\mu(t),t) && \text{(induction hypothesis)} \\ &= I(k,t+1) && \text{(lemma 80).} \end{aligned}$$

For part (2) we split cases exactly as for part (1).

- $\mu(t+1) = \mu(t)$. According to lemma 103, the stage below $\mu(t)$ is not updated in cycle t.

$$/ue^t_{\mu(t)+1}$$

Moreover, since by lemma 84 stage μ always has a real full bit ($rfull^t_{\mu(t)}$), from construction of the stall engine (part (3) of lemma 75) we conclude that stage $\mu(t)$ is not overwritten in cycle t.

$$/ue^t_{\mu(t)}$$

Finally, from lemma 101 we have

$$\mu(t) \geq R(t).$$

Thus, for stages $k \leq \mu(t)$ we derive:

$$\begin{aligned} i(k,t+1) &= i(k,t) && \text{(definition)} \\ &= I(\mu(t),t) && \text{(induction hypothesis)} \\ &= I(\mu(t),t+1) && \text{(lemma 80)} \\ &= I(\mu(t+1),t+1). \end{aligned}$$

- $\mu(t+1) = \mu(t)+1$. According to lemmas 103 and 104, stage $\mu(t)+1$ is updated in cycle t.

$$ue^t_{\mu(t)+1}$$

From lemma 101 we know that the latter stage is below R in cycle t.

$$\mu(t)+1 > R(t).$$

To complete the induction step, for stage $k = \mu(t)+1$ we derive:

$$\begin{aligned} i(k,t+1) &= i(k,t)+1 && \text{(definition)} \\ &= I(k,t)+1 && \text{(induction hypothesis)} \\ &= I(k,t+1) && \text{(lemma 80)} \\ &= I(\mu(t+1),t+1). \end{aligned}$$

For stages $k \leq \mu(t)$ we argue similarly to the case above:

$$\begin{aligned} i(k,t+1) &= i(k,t) && \text{(definition)} \\ &= I(\mu(t),t) && \text{(induction hypothesis)} \\ &= I(\mu(t)+1,t+1) && \text{(definition)} \\ &= I(\mu(t+1),t+1). \end{aligned}$$

- $\mu(t+1) = 0 \neq \mu(t)$. For stages $k < 0$ there is nothing to show. □

8.4.5 Uniqueness of Update Cycles

In Sect. 9.2.5 we must define the oracle inputs for instructions in circuit stages below μ, in cycles in which those instructions progress down in the pipeline. Therefore, we first must show that the latter cycles are unique. This last section is spent to derive that result. Below we argue that the value of μ resets to zero only after a rollback in stage μ.

Lemma 106. *Assume $\tilde{t} > t$ and $\mu(t) > 0$.*

$$\mu(\tilde{t}) = 0 \rightarrow \exists t' \in [t : \tilde{t} - 1] : \; rbr^{t'}_{\mu(t')}$$

Proof of lemma 106. Consider the minimum cycle $t_1 > t$ such that $\mu(t_1) = 0$. For cycle

$$t_0 = t_1 - 1 \geq t$$

we clearly have $\mu(t_0) > 0$, and according to lemma 103 we obtain

$$\forall t' \in [t : t_0 - 1] : /(rbr^{t'}_{\mu(t')} \wedge ue^{t'}_{\mu(t')+1})$$

since otherwise a contradiction follows. In cycle t_0, again according to lemma 103, we have

$$rbr^{t_0}_{\mu(t_0)} \wedge ue^{t_0}_{\mu(t_0)+1} = 1$$

and

$$\mu(t_0 + 1) = \mu(t_1) = 0.$$

Clearly

$$t_0 \in [t : t_1 - 1] \subseteq [t : \tilde{t} - 1]$$

and the claim follows. □

Next we show that the (non-zero) value of μ always eventually resets to zero.

Lemma 107. *Assume $\mu(t) > 0$.*

$$/rbr^{t}_{\mu(t)} \rightarrow \exists t' > t : \; rbr^{t'}_{\mu(t')}$$

Proof of lemma 107. By contradiction; assume that the following holds.

$$\forall t' > t : \; /rbr^{t'}_{\mu(t')} \tag{63}$$

First, we proceed to show the following.

$$\forall t' > t : \; \mu(t') > 0 \wedge I(\mu(t'), t') = I(\mu(t), t) \tag{64}$$

Proof of equation 64. In case $\mu(t') = 0$, from lemma 106 we immediately derive a contradiction.

$$\exists t'' \in [t : t' - 1] : \; rbr^{t''}_{\mu(t'')}$$

Therefore, we have

$$\forall t' > t : \; \mu(t') > 0$$

for the scheduling functions, by induction on θ, we proceed to show

$$\forall \theta \geq 0 : \; I(\mu(t + \theta), t + \theta) = I(\mu(t), t).$$

The base case holds trivially since for $\theta = 0$ there is nothing to show. For the induction step from θ to $\theta + 1$ we argue as follows. For $t'' = t + \theta$, in case the stage below $\mu(t'')$ is updated in cycle t'', we derive

$$\begin{aligned} I(\mu(t''+1), t''+1) &= I(\mu(t'') + 1, t'' + 1) && \text{(lemma 103.2)} \\ &= I(\mu(t''), t'') && \text{(definition)} \\ &= I(\mu(t), t) && \text{(induction hypothesis).} \end{aligned}$$

Otherwise, from lemma 84 we know that pipeline stage $\mu(t'')$ has a real full bit in cycle t''

$$rfull^{t''}_{\mu(t'')} = 1$$

which according to the construction of the stall engine (part (3) of lemma 75) implies

$$ue^{t''}_{\mu(t'')} = 0.$$

To complete the proof we argue similarly to the case above:

$$\begin{aligned} I(\mu(t''+1),t''+1) &= I(\mu(t''),t''+1) && \text{(lemma 103.1)} \\ &= I(\mu(t''),t'') && \text{(definition)} \\ &= I(\mu(t),t) && \text{(induction hypothesis).} \end{aligned}$$

□

Using the result above, we proceed to derive a contradiction (by definition of μ):

$$live(\mu(t),t).$$

From lemmas 98 and 82 we already have

$$\exists t'' \geq t: \; I(6,t'') = I(\mu(t),t) \wedge ue^{t''}_{6}$$

and (by definition of *live*) it remains to show

$$\forall \tilde{t} \in [t:t''] \; \forall \tilde{k} \in [\mu(t):5]: \; I(\tilde{k},\tilde{t}) = I(\mu(t),t) \rightarrow /rbr^{\tilde{t}}_{\tilde{k}}.$$

- For stage $\tilde{k} = \mu(t')$ directly from the assumption (equation 63) we derive

$$\forall t' > t: \; I(\tilde{k},t') = I(\mu(t),t) \rightarrow /rbr^{t'}_{\tilde{k}}.$$

- For stages $\tilde{k} > \mu(t')$, according to lemma 101, we have $/rbr^{t'}_{\tilde{k}}$, and therefore

$$\forall t' > t \; \forall \tilde{k} \in [\mu(t')+1:5]: \; I(\tilde{k},t') = I(\mu(t),t) \rightarrow /rbr^{t'}_{\tilde{k}}.$$

- Finally, for stages $\tilde{k} < \mu(t')$, we proceed to show

$$\forall t' > t \; \forall \tilde{k} \in [\mu(t):\mu(t')-1]: \; I(\tilde{k},t') = I(\mu(t),t) \rightarrow /rbr^{t'}_{\tilde{k}}.$$

  For maximal stage $k' < \mu(t')$ such that

$$rbr^{t'}_{k'} = 1$$

  from the hardware construction and equation 63 we have

$$\forall \ell \in [1:k']: \; rbr^{t'}_{\ell} \wedge /rbr^{t'}_{k'+1}$$

  and therefore, from the definition of the *misspec* signals we conclude

$$rfull^{t'}_{k'} = 1.$$

  Using the arguments above, for the scheduling functions of stages $\ell \leq k'$ we argue (by contrapositive):

$$\begin{aligned} I(\ell,t') &> I(k'+1,t') && \text{(lemma 81)} \\ &\geq I(\mu(t'),t') && \text{(lemma 81)} \\ &= I(\mu(t),t) && \text{(equation 64).} \end{aligned}$$

□

Using the result above we argue that every pipeline stage is updated infinitely often *below* stage μ.

Lemma 108.

$$/(ue_k^t \wedge k > \mu(t)) \;\rightarrow\; \exists t' > t : \; ue_k^{t'} \wedge k > \mu(t')$$

Proof of lemma 108. By case split on whether stage k is below μ in cycle t:

- $k > \mu(t)$. Consider minimum cycle $t' > t$ in which stage k is updated ($ue_k^{t'}$). Note, from lemma 99 we know that the latter cycle exists. Thus, we have
$$\forall \tilde{t} \in [t : t'-1] : /ue_k^{\tilde{t}} \wedge ue_k^{t'}.$$
We proceed to show
$$\forall \tilde{t} \in [t : t'] : \; k > \mu(\tilde{t}).$$
by induction on θ, where θ denotes the length of sub-interval
$$[t : t+\theta-1] \subseteq [t : t'].$$
The base case holds trivially since for $\theta = 0$ there is nothing to show. For the induction step from $\theta \leq t'-t$ to $\theta+1$ we argue as follows. For $t'' = t+\theta-1$, in case
$$\mu(t'') < k-1$$
using lemma 103 we obtain
$$\mu(t''+1) \leq k-1 < k.$$
Otherwise, in case
$$\mu(t'') = k-1$$
using part (1) of lemma 103 we have
$$\mu(t''+1) = k-1 < k$$
which completes the induction step, and thus the argument for this case.
- $k \leq \mu(t)$. From lemma 107 for some cycle $t' \geq t$ we know that stage $\mu(t')$ is rolled-back ($rbr_{\mu(t')}^{t'}$). According to lemma 101, we split cases on the value of
$$\mu(t') = R(t') \in \{2, 5\}.$$
Thus, in case $\mu(t') = 5$, from the hardware interconnect we derive that the memory stage is updated in t' ($ue_6^{t'}$), and using part (2) of lemma 103 we conclude
$$\mu(t'+1) = 0.$$
Otherwise, in case $\mu(t') = 2$, from lemma 99 for the minimum cycle $t'' \geq t'$ we have
$$\forall \tilde{t} \in [t' : t''-1] : /ue_3^{\tilde{t}} \wedge ue_3^{t''}.$$
Therefore, using part (1) of lemma 103 we derive
$$\forall \tilde{t} \in [t' : t''] : \; \mu(\tilde{t}) = 2.$$
Repeating the arguments from the proof of equation 53 we conclude
$$\forall \tilde{t} \in [t' : t''] : rbr_2^{\tilde{t}}$$
which by part (2) of lemma 103 implies
$$\mu(t''+1) = 0.$$
Therefore, in both cases, for some cycle $\bar{t} > t$ we have
$$k > \mu(\bar{t})$$
and the claim follows as in the case above for some $\bar{\bar{t}} > t$. □

Next we show that the hardware cycles in which instructions leave the pipeline stages below μ exist (lemma 109) and, moreover, unique (equation 65).

Lemma 109. *For stages $k \in [1:5]$ the following holds:*

$$\forall i\ \exists t:\ i = I(k,t) \wedge ue_k^t \wedge k > \mu(t)$$

Proof of lemma 109. By induction on index i of the instruction in stage k. For the induction base ($i = 0$) we argue as follows. From lemma 108 we know that stage k below μ is updated infinitely often. Consider the first such cycle t' in which stage k below μ is updated. Clearly, we have:

$$(\forall t \in [0:t'-1]: /(ue_k^t \wedge k > \mu(t))) \wedge ue_k^{t'} \wedge k > \mu(t').$$

For the scheduling function in stage k we conclude:

$$\begin{aligned} I(k,t') &= i(k,t') && \text{(lemma 105.1)} \\ &= i(k,0) && \text{(definition)} \\ &= 0 && \text{(definition).} \end{aligned}$$

For the induction step from i to $i+1$ we argue as follows. Directly from the induction hypothesis we know there is a cycle t such that

$$i = I(k,t) \wedge ue_k^t \wedge k > \mu(t).$$

Applying lemma 108 we also know there is a cycle $t' \geq t+1$:

$$ue_k^t \wedge k > \mu(t) \wedge (\forall \tilde{t} \in [t+1:t'-1]: /(ue_k^{\tilde{t}} \wedge k > \mu(\tilde{t}))) \wedge ue_k^{t'} \wedge k > \mu(t').$$

To complete the induction step, for the scheduling function in stage k we derive:

$$\begin{aligned} I(k,t') &= i(k,t') && \text{(lemma 105.1)} \\ &= i(k,t)+1 && \text{(definition)} \\ &= I(k,t)+1 && \text{(lemma 105.1)} \\ &= i+1. \end{aligned}$$

□

Finally, for stages $k \in [1:5]$ we argue

$$\forall i\ \exists!\, t:\ i = I(k,t) \wedge ue_k^t \wedge k > \mu(t) \tag{65}$$

Proof of equation 65. From lemma 109 for stages $k \in [1:5]$ we know

$$\forall i\ \exists t:\ i = I(k,t) \wedge ue_k^t \wedge k > \mu(t).$$

Assume stage $k \in [1:5]$ is updated below μ in cycles t and $t' > t$

$$\begin{aligned} &ue_k^t \wedge k > \mu(t) \\ &ue_k^{t'} \wedge k > \mu(t') \end{aligned}$$

with instruction

$$I(k,t) = I(k,t').$$

The contradiction immediately follows.

$$\begin{aligned} I(k,t') &= i(k,t') && \text{(lemma 105.1)} \\ &\geq i(k,t+1) && \text{(monotonicity)} \\ &= i(k,t)+1 && \text{(definition)} \\ &= I(k,t)+1 && \text{(lemma 105.1)} \end{aligned}$$

□

The latter uniqueness result completes the implementation of the pipelined machine. Correctness of the pipelined implementation is proven in the next chapter. (Note, in Chap. 9 we consider a *multi-core* processor, every core of which is a pipeline as constructed above in this chapter.)

9

Correctness of Pipelined Implementation

In Chap. 8 we constructed the pipelined processor which, as we prove in this chapter, implements the ISA specification from Sect. 9.1. In order to formalize our arguments, in Sect. 9.2 we as usual state the (simulation) theorem. After developing some more machinery in Sect. 9.3, we perform the induction steps for every part of the induction hypothesis. Most of the arguments in Sects. 9.4–9.8 are analogous to the arguments presented in the corresponding sections of Chap. 7. However, the interesting parts of course are Sect. 9.4, where we use the software conditions in order show that the guard conditions are respected by our pipelined implementation, and Sect. 9.5, where we apply the software conditions to prove correctness for the output of the instruction cache.

9.1 Multi-Core Specification

In this chapter we consider a simplified version of the multi-core machine described in [Sch13a]. Here we assume that *all* processor cores start running simultaneously after receiving a common *reset* interrupt, i.e., no *init* signals are required in the absence of *running* flags. As we already mentioned above, the external interrupts (other than *reset*) are tied to zero for all cores. Thus, we can keep using the same processor model as defined for the single-core machine in Sect. 2.1. The latter also allows us to keep the definition of inputs for the processor core steps unchanged.

9.1.1 Multi-Core Computation

Configuration

Configuration $mc \in K_{MC}$ of the multi-core machine with nested MMUs consists of p processors connected to one, shared byte-addressable memory:

- $mc.p : [0 : p-1] \rightarrow K_M$, and
- $mc.m : \mathbb{B}^{32} \rightarrow \mathbb{B}^8$.

In turn configuration of each processor q includes the following sub-components:

- $mc.p(q).(pc, dpc, ddpc) \in \mathbb{B}^{32}$ — three program counters,
- $mc.p(q).(gpr, spr) : \mathbb{B}^5 \rightarrow \mathbb{B}^{32}$ — general and special purpose register files, and
- $mc.p(q).tlb \subseteq K_{uwalk}$ — translation look-aside buffer.

Note, the initial configuration mc^0 of the multi-core system after reset must satisfy the following criteria: i) the program counters are initialized with proper values, ii) the first GPR register contains value zero, and iii) all maskable interrupts are masked.

$$\begin{aligned} mc^0.p(q).(pc, dpc, ddpc) &= (8_{32}, 4_{32}, 0_{32}) \\ mc^0.p(q).gpr(0_5) &= 0_{32} \\ mc^0.p(q).sr &= 0_{32} \end{aligned}$$

Moreover, the SPR registers have values which indicate the occurrence of *reset* on the previous step: i) the exception cause register has its first bit set to one and ii) both mode registers have their first bits set to zero.

$$\begin{aligned} mc^0.p(q).eca[0] &= 1 \\ mc^0.p(q).mode[0] &= 0 \\ mc^0.p(q).nmode[0] &= 0 \end{aligned}$$

Stepping Function

Every sequential global step number n is performed by the component and using the oracle input specified by the stepping function

$$s: \mathbb{N} \to \Sigma$$

where

$$\begin{aligned} \Sigma = \{core\} \times [0:p-1] \times \Sigma_{core} \cup \\ \{tlb\} \times [0:p-1] \times \Sigma_{tlb}. \end{aligned}$$

For convenience we introduce the following shorthands to select the fields of the stepping function value:

$$s(n) = s(n).(t,u,o)$$

where

- $.t$ gives the type of component which is stepped; Obviously, we have
$$s(n).t \in \{core, tlb\}.$$
- $.u$ gives the number of unit which is stepped; Clearly
$$s(n).u \in [0:p-1].$$
- $.o$ gives the oracle input for the step as defined in the components specification; For inputs we have
$$s(n).o \in \Sigma_{core} \cup \Sigma_{tlb}.$$

Moreover, we partition the oracle inputs for the processor core steps further using the shorthands below. The meaning of particular fields is intuitive and matches the specification from Sect. 3.3.4.

$$s(n).t = core \ \to \ s(n).o = s(n).o.(w_I, w_E, eev) \in \Sigma_{core}$$

Semantics

Below we formalize the semantics of the multi-core MIPS machine with nested address translation by defining the transition function δ_{MC}, which specifies for any global step number n the next state configuration mc^{n+1}. Depending on which component makes a step we distinguish two cases:

- $s(n).(t,u) = (core, q)$ — step n is performed by the core of processor q. For the next step configuration of processor q and the shared memory component we have:
$$mc^{n+1}.(p(q),m) = \delta_M(mc^n.(p(q),m), s(n).o)$$
where δ_M denotes the transition function of the single-core MIPS with the nested address translation, which we specified previously in Sect. 3.3.4.

- $s(n).(t,u) = (tlb,q)$ — step n is performed by the TLB of processor q. For the next step configuration of the TLB component of processor q we have:

  $$mc^{n+1}.tlb = \delta_T(mc^n.(p(q),m),s(n).o)$$

  where δ_T is the transition function of the TLB component, specified in Sect. 3.3.1. Note that the remaining components of processor q as well as the shared memory component do not change at this step:

  $$\begin{aligned} Z \neq tlb \;\to\; mc^{n+1}.p(q).Z &= mc^n.p(q).Z \\ mc^{n+1}.m &= mc^n.m. \end{aligned}$$

In both cases all other processors do not change at step n:

$$q' \neq q \;\to\; mc^{n+1}.p(q') = mc^n.p(q').$$

Guard Conditions

Using notation from Sect. 3.3 we introduce the guard conditions for the multi-core machine with almost no effort. Thus, for multi-core configuration $mc \in K_{MC}$ and oracle input $\sigma \in \Sigma$ we define a single predicate Γ which covers all conditions restricting the computational steps:

$$\Gamma(mc,\sigma) \equiv \begin{cases} \Phi(mc.p(\sigma.u),\sigma.o) & \sigma.t = core \\ T(mc.p(\sigma.u),\sigma.o) & \sigma.t = tlb. \end{cases}$$

In case the guard conditions hold for all ISA steps before step n, we abbreviate

$$\Gamma^n \;\equiv\; \forall m < n : \Gamma(mc^m, s(m)).$$

9.1.2 Processor Local Computations

We begin with the very basic definitions. As usual, for processor q we define two auxiliary functions to formalize the processor core step sequence

$$\begin{aligned} pseq(q,0) &= \min\{n \mid s(n).(t,u) = (core,q)\} \\ pseq(q,i+1) &= \min\{n \mid s(n).(t,u) = (core,q) \wedge n > pseq(q,i)\} \end{aligned}$$

and the instruction count

$$\begin{aligned} ic(q,0) &= 0 \\ ic(q,n+1) &= \begin{cases} ic(q,n)+1 & s(n).(t,u) = (core,q) \\ ic(q,n) & \text{otherwise.} \end{cases} \end{aligned}$$

Two simple lemmas follow directly from the definitions above.

Lemma 110.

$$s(n).(t,u) = (core,q) \;\to\; pseq(q,ic(q,n)) = n$$

Lemma 111.

$$pseq(q,ic(q,n)) \geq n$$

Their proofs literally follow the arguments from Sect. 7.2.1, where the counter parts of these two lemmas were proven for the single core (sequential) machine. For technical reasons we define

$$pseq(q,-1) = -1 \tag{66}$$

and show the following result.

Lemma 112.

$$pseq(q, ic(q,n) - 1) < n$$

Proof of lemma 112. By induction on the number of steps n. For the base case ($n = 0$) we argue as follows.

$$\begin{aligned} pseq(q, ic(q,0) - 1) &= pseq(q, -1) \qquad \text{(definition)} \\ &= -1 \qquad \text{(equation 66)} \\ &< 0. \end{aligned}$$

For the induction step from n to $n+1$ we split cases on the type of the step performed.

- $s(n).(t,u) = (core, q)$. In case step n is a core step of processor q, we have:

$$\begin{aligned} pseq(q, ic(q,n+1) - 1) &= pseq(q, ic(q,n)) \qquad \text{(definition)} \\ &= n \qquad \text{(lemma 110)} \\ &< n+1. \end{aligned}$$

- $s(n).(t,u) \neq (core, q)$. Otherwise, we argue using the induction hypothesis:

$$\begin{aligned} pseq(q, ic(q,n+1) - 1) &= pseq(q, ic(q,n) - 1) \qquad \text{(definition)} \\ &< n \qquad \text{(induction hypothesis)} \\ &< n+1. \end{aligned}$$

□

Next we establish a counterpart of lemma 66 for the multi-core machine.

Lemma 113.

$$\forall n : \; m \in [n : pseq(q, ic(q,n)) - 1] \;\rightarrow\; s(m).(t,u) \neq (core, q)$$

Proof of lemma 113. We prove the statement by induction on ℓ, where ℓ denotes the length of sub-interval

$$[n : n + \ell - 1] \subseteq [n : pseq(q, ic(q,n)) - 1].$$

The base case holds trivially since for $\ell = 0$ there is nothing to show. For the induction step from $\ell - 1$ to

$$\ell \leq pseq(q, ic(q,n)) - n$$

we argue by contradiction as follows. Assume that $n + \ell$ is a core step of processor q:

$$s(n+\ell).(t,u) = (core, q).$$

Then a contradiction follows simply from the monotonicity of functions *ic* and *pseq*:

$$\begin{aligned} n + \ell &= pseq(q, ic(q, n+\ell)) \qquad \text{(lemma 110)} \\ &\geq pseq(q, ic(q,n)) \\ &> pseq(q, ic(q,n)) - 1. \end{aligned}$$

□

In the following lemma we capture a property of *pseq*, namely that the processors do not perform steps in configurations apart from the configurations given by *pseq*.

Lemma 114.

$$m \in [pseq(q,i) + 1 : pseq(q, i+1) - 1] \;\rightarrow\; s(m).(t,u) \neq (core, q)$$

Proof of lemma 114. By contradiction. Assume there is n such that:

$$n \in [pseq(q,i)+1 : pseq(q,i+1)-1] \wedge s(n).(t,u) = (core,q).$$

Then for the instruction index

$$i_n = ic(q,n)$$

using lemma 110 we conclude:

$$pseq(q,i_n) = n.$$

From the monotonicity of function *pseq* we derive a contradiction:

$$pseq(q,i) < pseq(q,i_n) < pseq(q,i+1)$$

gives

$$i < i_n < i+1. \qquad \square$$

The next lemma is a counterpart of lemmas 9.9 from [KMP14].

Lemma 115.

$$mc^n.p(q).core = mc^{pseq(q,ic(q,n))}.p(q).core$$

Proof of lemma 115. By induction on n. For the base case we trivially obtain

$$\begin{aligned} mc^{pseq(q,ic(q,0))}.p(q).core &= mc^{pseq(q,0)}.p(q).core && \text{(definition)} \\ &= mc^0.p(q).core && \text{(lemma 113).} \end{aligned}$$

For the induction step from n to $(n+1)$ we split cases on whether step n is performed by processor q or not.

- $s(n).(t,u) = (core,q)$:

$$\begin{aligned} mc^{pseq(q,ic(q,n+1))}.p(q).core &= mc^{pseq(q,ic(q,n)+1)}.p(q).core && \text{(definition)} \\ &= mc^{pseq(q,ic(q,n))+1}.p(q).core && \text{(lemma 114)} \\ &= mc^{n+1}.p(q).core && \text{(induction hypothesis).} \end{aligned}$$

- $s(n).(t,u) \neq (core,q)$:

$$\begin{aligned} mc^{pseq(q,ic(q,n+1))}.p(q).core &= mc^{pseq(q,ic(q,n))}.p(q).core && \text{(definition)} \\ &= mc^n.p(q).core && \text{(induction hypothesis)} \\ &= mc^{n+1}.p(q).core && \text{(definition).} \end{aligned} \qquad \square$$

As always, signals Z on processor q of a local configuration i are abbreviated as:

$$Z_\sigma^{q,i} = Z(mc^{pseq(q,i)}.p(q), s(pseq(q,i)).o).$$

For instance, we abbreviate the walks passed to the specification machine by

$$\begin{aligned} w_{I\ \sigma}^{q,i} &= s(pseq(q,i)).o.w_I \\ w_{E\ \sigma}^{q,i} &= s(pseq(q,i)).o.w_E \end{aligned}$$

whereas for the external event signals passed we introduce in a natural way:

$$eev_\sigma^{q,i} = s(pseq(q,i)).o.eev.$$

Similarly, we abbreviate the used predicates in global steps as

$$used(Z,n) = used(Z, mc^n, s(n))$$

and in processor local steps as

$$used(Z)_\sigma^{q,i} = used(Z, mc^{pseq(q,i)}.p(q), s(pseq(q,i)).o).$$

9.1.3 Accessing Memory

In [KMP14] the type of the access was an abbreviation:

$$.type = .(r, w, cas, f).$$

The type component of an access is always used. Using the notation above we define for $acc \in K_{acc}$, any step number n, processor q, and instruction index i:

$$used(acc.type, n) = 1$$
$$used(acc.type)_{\sigma}^{q,i} = 1.$$

For those components of accesses which are never used (e.g., *.data* components of instruction accesses) the corresponding definitions are omitted below.

Address Translation

For translation of the instruction and effective address we introduce the access

$$tacc: \ \mathbb{N} \to K_{acc}$$

(for translation access) which we define as follows. In case a TLB step is performed

$$s(n).t = tlb$$

we specify the translation access using notation from Sect. 3.3.2.

$$tacc(n) = tacc(s(n).o)$$

Otherwise, we simply specify a void access.

$$tacc(n).type = 0000$$

For convenience we abbreviate the output of the translation access in ISA step n as

$$tmout(n) = dataout(\ell(mc^{n}.m), tacc(n)).$$

For the used predicates we obviously define:

$$used(tacc.a, n) = /void(tacc(n)).$$

Instruction Fetch

For fetching instructions we update the definition of the instruction access to match the new specification. Now instructions are fetched from the pma_I. In contrast to the translation access above, the instruction access

$$iacc: \ K_C \times \Sigma_{core} \to K_{acc}$$

is defined by the local processor configuration. In the absence of interrupts of level 4 or lower

$$il_{\sigma}^{q,i} > 4$$

we specify the instruction access to be a read of the memory line containing the instruction word.

$$iacc_{\sigma}^{q,i}.a = pma_{I}{}_{\sigma}^{q,i}.l$$
$$iacc_{\sigma}^{q,i}.type = 1000$$

Otherwise ($il_{\sigma}^{q,i} \leq 4$), a void access is specified.

$$iacc_{\sigma}^{q,i}.type = 0000$$

For convenience we abbreviate as usual,

$$iacc(q, i) = iacc_{\sigma}^{q,i}.$$

For the used predicates we define:

$$used(iacc.a)_{\sigma}^{q,i} = /void(iacc(q, i)).$$

Data Access

We update the definition of the data access in the same way we updated the definition of the instruction access above. Formally the data access

$$dacc: \ K_C \times \Sigma_{core} \to K_{acc}$$

is defined by the local processor configuration. In the absence of JISR on a memory operation

$$mop_\sigma^{q,i} \wedge /\, jisr_\sigma^{q,i}$$

we specify a non-void access based on the following signals.

$$\begin{aligned}
dacc_\sigma^{q,i}.a &= pma_{E\ \sigma}^{q,i}.l \\
dacc_\sigma^{q,i}.data &= dmin_\sigma^{q,i} \\
dacc_\sigma^{q,i}.cdata &= cdata_\sigma^{q,i} \\
dacc_\sigma^{q,i}.bw &= bw_\sigma^{q,i} \\
dacc_\sigma^{q,i}.type &= (l,s,cas,0)_\sigma^{q,i}
\end{aligned}$$

Otherwise, we simply specify a void access.

$$dacc_\sigma^{q,i}.type = 0000$$

As usual we abbreviate

$$dacc(q,i) = dacc_\sigma^{q,i}.$$

Finally, for the used predicates we define:

$$\begin{aligned}
used(dacc.a)_\sigma^{q,i} &= /void(dacc(q,i)) \\
used(dacc.cdata)_\sigma^{q,i} &= dacc(q,i).cas \\
used(dacc.data)_\sigma^{q,i} &\equiv dacc(q,i).(w,cas) \neq 0_2 \\
used(dacc.bw)_\sigma^{q,i} &= used(dacc.data)_\sigma^{q,i}.
\end{aligned}$$

9.2 Multi-Core Processor

Following the specification from Sect. 9.1, in a completely straightforward way we assemble a multi-core processor by putting p pipelined cores (as constructed in Chap. 8) in parallel. Clearly, the total number of caches involved increases to $4p$. Everywhere in the remainder of this chapter we use indices

$$q \in [0 : p-1]$$

to iterate over processors $h_\pi.p(q)$ of the multi-core machine h_π. Of course, the four caches connected to processor number q are chosen simply as follows.

$$\begin{aligned}
itca_\pi^q &= h_\pi.ca(4q) \\
ica_\pi^q &= h_\pi.ca(4q+1) \\
dtca_\pi^q &= h_\pi.ca(4q+2) \\
dca_\pi^q &= h_\pi.ca(4q+3)
\end{aligned}$$

Note, for the sake of simplicity we may omit the processor indices (q) where possible.

9.2.1 Stepping of Components

In this section we specify i) which components perform steps in a given hardware cycle and ii) the order in which these steps are performed. Recall, in Chap. 7 we introduced predicates $cstep$ and $tstep_Y$ to specify resp. steps of the processor core and TLBs of the sequential single-core machine. We update these predicates for use in the multi-core machine. Thus, processor q performs a core step in cycle t in case the memory stage on processor q is updated in cycle t.

$$cstep(q,t) \;\equiv\; ue_6^{q,t}$$

Processor q performs a step of TLB $Y \in \{I,E\}$ in cycle t in case the corresponding MMU on processor q has generated a step in cycle t.

$$tstep_Y(q,t) \;\equiv\; tadd(h_\pi^t.p(q).mmu_Y)$$

The processor cores which are stepped in cycle t are collected into the set

$$PS(t) = \{q \mid cstep(q,t)\}$$

whereas the TLBs which are stepped in cycle t — into the sets

$$TS_Y(t) = \{q \mid tstep_Y(q,t)\}$$

depending on the MMU (mmu_I or mmu_E) which generated the corresponding step. The total number of steps performed by all components in cycle t can be easily expressed using the definitions above; we denote it by

$$ns(t) = \#PS(t) + \#TS_I(t) + \#TS_E(t).$$

At this point we can write down the usual definition of the total number of steps performed by all components *before* cycle t:

$$\begin{aligned} NS(0) &= 0 \\ NS(t+1) &= NS(t) + ns(t). \end{aligned}$$

For convenience, we also collect the numbers of steps performed within cycle t into the set

$$CS(t) = [NS(t) : NS(t+1) - 1].$$

Note, ordering of the steps performed within cycle t, and therefore the global ordering of steps, is in general not arbitrary. Broadly speaking, due to specifics of the control mechanisms implemented in hardware, a certain ordering of steps can be enforced as the only possible in particular situations (cycles). In our design we implemented control mechanisms in a way that allows us to step the components which perform memory accesses prior to the remaining components stepped. On the other hand, in order to make the proofs independent of particular implementation of the cache memory system, components accessing memory are stepped in the sequential order in which the memory system in use orders the latter accesses.

In order to filter out accesses of the MMUs which do not generate steps (for instance, due to possible abortions), we introduce some more notation. Three sets below accumulate accesses generated by the MMUs (S_I and S_E) as well as accesses generated by processors (S_D).

$$\begin{aligned} S_I(t) &= \{(i,k) \in A(t) \mid (i \bmod 4) = 0 \wedge \lfloor i/4 \rfloor \in TS_I(t)\} \\ S_E(t) &= \{(i,k) \in A(t) \mid (i \bmod 4) = 2 \wedge \lfloor i/4 \rfloor \in TS_E(t)\} \\ S_D(t) &= \{(i,k) \in A(t) \mid (i \bmod 4) = 3 \wedge \lfloor i/4 \rfloor \in PS(t)\} \end{aligned}$$

Note, only those accesses are included under the condition that the corresponding unit actually performs a step in cycle t. By definition we obviously have

$$S(t) = S_I(t) \cup S_E(t) \cup S_D(t) \subseteq A(t).$$

As usual, we abbreviate the total number of the stepping accesses by

$$sa(t) = \#S(t).$$

Analogous to Sect. 8.3.5, for $y < sa(t)$ we define:

$$\begin{aligned} z(y,0) &= \min\{n \mid NE(t) + n \in seq(S(t))\} \\ z(y,t) &= \min\{n \mid NE(t) + n \in seq(S(t)) \wedge n > z(y-1,t)\}. \end{aligned}$$

Using function z for indices

$$y \in [0 : sa(t) - 1]$$

we can easily extract from sequence acc' the *sequence of stepping accesses* $zacc'_t$:

$$zacc'_t[y] = acc'[NE(t) + z(y,t)].$$

Before we define the stepping order, we introduce one more technicality, which helps us to distinguish between possibly two different TLB steps performed by the same processor in the same hardware cycle. We extend the stepping function to include another field $.g$ that gives the component which *generates* the latter step. Formally, for global step n we specify

$$s(n).g \in \{proc, mmu_I, mmu_E\}.$$

Now we can specify the order in which components are stepped. Below we split cases on the value of $y < ns(t)$.

- $y < sa(t)$. Components performing memory accesses are stepped in the order in which the corresponding memory accesses are performed.

$$s(NS(t)+y).(g,u) = (proc,q) \leftrightarrow \exists k : NE(t) + z(y,t) = seq(4q+3,k) \tag{67}$$

$$s(NS(t)+y).(g,u) = (mmu_Y,q) \leftrightarrow \exists k : NE(t) + z(y,t) = seq(4q+2\mathbb{1}_{\{Y=E\}},k) \tag{68}$$

- $y \geq sa(t)$. Components which do not perform memory accesses are stepped in an arbitrary order. Each such component is stepped exactly once. Formally

$$s(NS(t) + [sa(t) : ns(t)-1]).(g,u) = (proc, Q_D(t)) \cup \bigcup_Y (mmu_Y, Q_Y(t))$$

  where sets $Q_D(t)$ and $Q_Y(t)$ collect the processor indices resp. of the cores and MMUs which generate steps in cycle t and perform no memory accesses.

$$\begin{aligned} Q_D(t) &= \{q \mid \forall y < sa(t) : s(NS(t)+y).(g,u) \neq (proc,q)\} \cap PS(t) \\ Q_Y(t) &= \{q \mid \forall y < sa(t) : s(NS(t)+y).(g,u) \neq (mmu_Y,q)\} \cap TS_Y(t) \end{aligned}$$

Finally, we derive the step types based on types of the components which generate the corresponding steps in a natural way.

$$s(n).t = \begin{cases} core & s(n).g = proc \\ tlb & \text{otherwise} \end{cases}$$

9.2.2 Software Conditions

Software conditions formulated for the basic MIPS machine from Chap. 2 (Sect. 2.2.2) are no longer valid for the pipelined multi-core machine and updated as follows. First, these conditions are relaxed in the part that the addresses of memory accesses, both for instruction fetch and memory operations, no longer have to be aligned. Now the latter conditions are tested by hardware, and whenever violated, corresponding misalignment interrupts are raised. This guarantees that the memory is still accessed only at aligned addresses.

On the other hand, we have to strengthen the software conditions in order to handle the problem which occurs in pipelined machines due to self-modifying codes. Namely, instruction i fetched by the instruction fetch stage at address a is outdated (incorrect) if the memory at address a is overwritten by any of the lower stages of the (instruction) local pipeline, or simply by any non-local pipeline of the multi-core processor.[1] In [KMP14] this problem was avoided by introducing two disjoint regions for code and data, and restricting the code region to be read only, which effectively forbids any self-modifying codes. In this thesis the self-modifying codes are allowed under the following software condition. For instruction i executed on processor q we require that its address ($pma_{I\ \sigma}^{q,i}$) is not written in ISA steps after execution of instruction $(i-5)$ and before execution of instruction $(i+1)$ on processor q.

$$\begin{aligned} SC_{code}(q,i) \equiv \forall n \in [pseq(q,i-5)+1 : pseq(q,i+1)-1] :& \\ s(n).(t,u) = (core,q') \wedge write(mc^n.p(q'),s(n).o) \rightarrow& \\ pma_E(mc^n.p(q'),s(n).o).l \neq pma_{I\ \sigma}^{q,i}.l& \end{aligned} \tag{69}$$

Note, in the software condition above we expose the depth of the pipelines used in the multi-core machine. For these pipelines we know that local instruction i is not fetched earlier than local instruction $(i-5)$ is executed. Therefore, in Sect. 9.5.3 we can show that the address of instruction i fetched on processor q is not modified while instruction i progresses through the local pipeline, neither by instructions in the lower stages on processor q nor by the other processors.

Finally, the software condition forbidding store or CAS operations on ROM remains. Below we simply reformulate the latter condition to fit description of the multi-core machine.

$$SC_{ROM}(q,i) \ \equiv\ write_\sigma^{q,i} \ \rightarrow\ \langle pma_{E\ \sigma}^{q,i}.l\rangle \geq 2^r \tag{70}$$

Formally for instruction i executed on processor q we define the software conditions to be follows.

$$SC(q,i) \ \equiv\ SC_{code}(q,i) \wedge SC_{ROM}(q,i)$$

Recall from Sect. 2.2.2 that the software conditions are assumed to hold only if the corresponding guard conditions are respected. For the multi-core machine we formalize the latter assumption as follows: the software conditions for instruction fetch (SC_{code}) are not violated by instruction i executed on processor q if the guard conditions are respected by all ISA steps until $pseq(q,i)$ and by the fetch in step $pseq(q,i)$. Using lemma 110 we derive the following.

$$s(n).(t,u) = (core,q) \wedge \Gamma^n \wedge \Phi_I(mc^n.p(q),s(n).o) \ \rightarrow\ SC_{code}(q,ic(q,n)) \tag{71}$$

Clearly, all software conditions are not violated by instruction executed on processor q if the guard conditions are respected by all ISA steps until $pseq(q,i)$ and by step $pseq(q,i)$. Again, using lemma 110 we summarize the latter arguments as follows.

$$s(n).(t,u) = (core,q) \wedge \Gamma^{n+1} \ \rightarrow\ SC(q,ic(q,n)) \tag{72}$$

9.2.3 Speculation Stage

Instructions in the upper pipeline stages (above the memory stage) not necessarily obey the software conditions that we formally described in Sect. 9.2.2 above. In cases the latter instructions disobey the software conditions, the corresponding *instruction stages* obviously cannot contain the correct data, and should be excluded from the correctness statement we formulate in Sect. 9.2.4. Below we introduce some more machinery which helps us to handle the aforementioned cases. Note, for convenience processor indices are omitted in the scope of this section.

All truly full stages above the memory stage containing the data of instructions which disobey the software conditions for instruction fetch are collected into the set

[1] Note, though in single-core pipelined machines some forwarding circuits can be added to keep track of the addresses written in the lower pipeline stages, in the multi-core machines this would require adding some extra mechanisms for the inter-processor communication. Up to our knowledge, modern processors do not implement such mechanisms.

$$S_{soft}(t) = \{k \in [1:5] \mid rfull_k^t \wedge /SC_{code}(I(k,t)-1)\}.$$

This allows us to determine the "lowest" pipeline stage containing the relevant data; it is called the *software speculation stage* and formally defined as follows:

$$\Sigma_{soft}(t) = \begin{cases} \max\ S_{soft}(t) & S_{soft}(t) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Additionally, in the presence of misspeculation we clearly cannot claim correctness of the pipeline stages which are the subject for rollbacks. Thus, all truly full stages containing the data of instructions executed with JISR (above the memory stage) and *eret* instructions (in stages 1 and 2) are collected resp. into the following sets.

$$\begin{aligned} S_{jisr}(t) &= \{k \in [1:5] \mid rfull_k^t \wedge jisr_\sigma^{I(k,t)-1}\} \\ S_{eret}(t) &= \{k \in [1:2] \mid rfull_k^t \wedge eret_\sigma^{I(k,t)-1}\} \end{aligned}$$

Note, in the definitions above we do not distinguish between legal and illegal *eret* instructions. The reason being is that any illegal instruction anyway triggers a JISR in the ISA computation. Using the sets above we introduce yet another speculation stage

$$\Sigma(t) = \max\{\Sigma_{jisr}(t), \Sigma_{eret}(t)\}$$

where

$$\Sigma_{jisr}(t) = \begin{cases} \max\ S_{jisr}(t) & S_{jisr}(t) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

and

$$\Sigma_{eret}(t) = \begin{cases} \max\ S_{eret}(t) & S_{eret}(t) \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

The latter allows us to determine the lowest pipeline stage containing the data of instruction that is executed, both in hardware and in software, without being rolled-back; it is called the *hardware speculation stage* and defined, clearly, as follows:

$$\Sigma_{hard}(t) = \max\{\Sigma(t), \mu(t)\}.$$

For convenience, we introduce the following shorthands.

$$\begin{aligned} \hat{\Sigma}(t) &= \max\{\Sigma_{soft}(t), \Sigma_{hard}(t)\} \\ \vec{\Sigma}(t) &= (\Sigma_{soft}(t), \Sigma_{hard}(t)) \end{aligned}$$

9.2.4 Induction Hypothesis

Essentially, the simulation theorem proven in this chapter is a counterpart of the corresponding theorem proven in the last chapter of [KMP14]. Here the latter result is extended to handle the (internal) interrupts and two more pipeline stages, added to support virtual address translation. Since the form of the simulation theorem does not change, to ease the presentation we proceed directly to the induction hypothesis.

The hypothesis as usual consists of several parts, which we introduce below in separate sections:

i) fulfillment of guard conditions imposed on the constructed ISA computation,
ii) embedding of the line-addressable abstraction of the hardware memory (cache memory system) into the byte-addressable memory of the ISA,
iii) presence of walks and walk compositions accumulated by the hardware (MMUs) resp. in the TLB and constructed TLB of the ISA,
iv) coupling of the pipeline registers; includes couplings of both, visible and invisible pipeline registers, resp. with the data structures and signals of the ISA,
v) coupling of the ghost walk registers (with the signals of the ISA), and
vi) invariants on contents of the ghost walk registers.

Guard Conditions

First, we require that steps of the constructed ISA computation respect their corresponding guard conditions.

$$\Gamma^{NS(t)}$$

Simulation of Memory

For abstraction of the cache memory system we have:

$$m(h_\pi^t) = \ell(mc^{NS(t)}.m).$$

Simulation of TLBs

Recall, each processor of the multi-core machine contains two MMUs, one for translation of the instruction addresses and one for translation of the effective addresses:

$$h_\pi.p(q).mmu_I \quad \text{and} \quad h_\pi.p(q).mmu_E.$$

We abbreviate the mmu_Y of processor q as

$$mmu_Y^q = h_\pi.p(q).mmu_Y.$$

Using the simulation relation for TLBs, introduced in Sect. 5.2.2, for MMUs of processor q we have:

$$sim_{tlb}(mmu_Y^{q,t}, mc^{NS(t)}.p(q)).$$

Simulation of Pipeline Registers

Let i be the index of instruction executed in stage k of processor q in cycle t

$$i = I(q,k,t)$$

and R be a pipeline register (configuration component) in stage k of processor q.

$$R \in reg(k)$$

As we stated previously in Sect. 8.2.3, we claim correctness only for instructions which are executed *without being rolled-back*, i.e., instructions in the live stages. Therefore, we additionally assume liveness of the instruction used for simulation of certain pipeline registers. Thus, for simulation of the visible registers in stage k we assume liveness of instruction i, i.e., $live(k,t)$ on processor q, which according to the definitions is equivalent to

$$k > \mu(q,t).$$

For the invisible registers in stage k we analogously assume liveness of the related instruction, i.e., liveness of instruction

$$\begin{aligned} i-1 &= I(q,k,t)-1 \\ &= I(q,k+1,t) \qquad \text{(lemma 79)} \end{aligned}$$

on processor q, or equivalently

$$k+1 > \mu(q,t).$$

Analogous to the arguments above, for visible registers in stage k we assume that instruction i is executed no later than the execution of $jisr$ or $eret$. Formally, in case

$$\Sigma(q,t) = \sigma > 0$$

by definition we have $jisr_\sigma^{q,j}$ or $eret_\sigma^{q,j}$ for

Table 12: Local invisible registers in various pipeline stages

stage #	stage name	invisible registers
1	IT	$w_I.1$, $ca.1_\pi[\mathtt{gf}:0]$, $pmaI.1_\pi$
2	IF	$w_I.2$, $ca.2_\pi[\mathtt{gf}:0]$
3	ID	$w_I.3$, $ca.3_\pi[\mathtt{gf}:0]$

...

$$\begin{aligned} j &= I(q,\sigma,t)-1 \\ &= I(q,\sigma+1,t) \qquad \text{(lemma 79).} \end{aligned}$$

As we assume $i \le j$ on processor q, using lemma 81 we conclude

$$k \ge \sigma + 1$$

or equivalently

$$k > \Sigma(q,t).$$

For the invisible registers in stage k, repeating the arguments above for the related instruction, i.e., instruction

$$i-1 = I(q,k+1,t)$$

on processor q, we analogously derive

$$k+1 > \Sigma(t).$$

Finally, for the visible registers in stage k we assume that no instructions violating the software conditions were executed before execution of instruction i. Formally, if

$$\Sigma_{soft}(q,t) = \sigma > 0$$

by definition we have $/SC_{code}(q,j)$ for

$$j = I(q,\sigma+1,t).$$

Therefore, assuming $i \le j$ on processor q, we exactly as above derive

$$k > \Sigma_{soft}(q,t).$$

For the invisible registers in stage k, which contain the data (*after* execution) of the related instruction, i.e., data of instruction $i-1$ on processor q, we must consider whether their content actually depends on instruction $i-1$ or not. For that reason we distinguish between the *local invisible* (Table 12) and *non-local invisible* registers. As the name suggests, content of local invisible registers is determined purely by the processor (local) state, and does not depend on instruction word, fetched from the memory.

Therefore, for the local invisible registers in stage k we can assume, just as for the visible registers, that no instructions violating the software conditions were executed *before* execution of the related instruction, i.e.,

$$\begin{aligned} i-1 &= I(q,k+1,t) \\ &\le I(q,\sigma+1,t) = j \end{aligned}$$

on processor q. Repeating the arguments presented above, for local invisible registers we derive

$$k+1 > \Sigma_{soft}(q,t).$$

On the other hand, for the non-local invisible registers in stage k we assume that no instructions violating the software conditions were executed *up to* the execution of the related instruction, i.e.,

$$\begin{aligned} i-1 &= I(q,k+1,t) \\ &< I(q,\sigma+1,t) = j \end{aligned}$$

on processor q. Following the same arguments as those presented above, for non-local invisible registers we obtain

$$k > \Sigma_{soft}(q,t).$$

Summarizing the arguments above, depending on the type of register R, we require:

- for a visible register
$$\hat{\Sigma}(q,t) < k \ \rightarrow\ R_{\pi}^{q,t} = R_{\sigma}^{q,i},$$
- for a local invisible register
$$\hat{\Sigma}(q,t) \le k \wedge rfull_k^{q,t} \wedge used(R)_{\sigma}^{q,i-1} \ \rightarrow\ R_{\pi}^{q,t} = R_{\sigma}^{q,i-1},$$
- for a non-local invisible register
$$\vec{\Sigma}(q,t) \le (k-1,k) \wedge rfull_k^{q,t} \wedge used(R)_{\sigma}^{q,i-1} \ \rightarrow\ R_{\pi}^{q,t} = R_{\sigma}^{q,i-1}.$$

Simulation of Ghost Registers

Below we abbreviate the ghost walk registers of processor q in cycle t as

$$\begin{aligned} w_I.k^{q,t} &= h_{\pi}^{t}.p(q).w_I.k \\ w_E.5^{q,t} &= h_{\pi}^{t}.p(q).w_E.5 \end{aligned}$$

and require, as for the ordinary pipeline (invisible) registers, the following to hold.

$$\begin{aligned} \hat{\Sigma}(q,t) \le k \wedge rfull_k^{q,t} \wedge used(w_I)_{\sigma}^{q,i-1} &\ \rightarrow\ w_I.k^{q,t} = w_{I\ \sigma}^{q,i-1} \\ \vec{\Sigma}(q,t) \le (4,5) \wedge rfull_5^{q,t} \wedge used(w_E)_{\sigma}^{q,i-1} &\ \rightarrow\ w_E.5^{q,t} = w_{E\ \sigma}^{q,i-1} \end{aligned}$$

Invariants

In addition we require for the ghost walk registers two more things. First, we require that the ghost registers on processor q are contained, depending on the execution mode of the pipeline, either in the (specification) TLB or in the constructed TLB.

Invariant 10.

$$\hat{\Sigma}(q,t) \le k \wedge rfull_k^{q,t} \wedge used(w_I)_{\sigma}^{q,i-1} \ \rightarrow\ w_I.k^{q,t} \in \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases}$$

$$\vec{\Sigma}(q,t) \le (4,5) \wedge rfull_5^{q,t} \wedge used(w_E)_{\sigma}^{q,i-1} \ \rightarrow\ w_E.5^{q,t} \in \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases}$$

Also these ghost registers have to match the translation requests of the instructions executed on processor q in the corresponding pipeline stages.

Invariant 11.

$$\begin{aligned} \hat{\Sigma}(q,t) \le k \wedge rfull_k^{q,t} \wedge used(w_I)_{\sigma}^{q,i-1} &\ \rightarrow\ match(trq_{I\ \sigma}^{q,i-1}, w_I.k^{q,t}) \\ \vec{\Sigma}(q,t) \le (4,5) \wedge rfull_5^{q,t} \wedge used(w_E)_{\sigma}^{q,i-1} &\ \rightarrow\ match(trq_{E\ \sigma}^{q,i-1}, w_E.5^{q,t}) \end{aligned}$$

Induction Base

As usual, after the hardware reset the simulation of the ISA data structures is obtained via extracting an appropriate ISA configuration mc^0 from the hardware configuration h^0. In this regard for simulation of the ISA memory it suffices to specify

$$\ell(mc^0.m) = m(h^0_\pi).$$

For the program counters and register files on all processors q we obtain analogously:

$$\begin{aligned} mc^0.p(q).(pc, dpc, ddpc) &= h^0_\pi.p(q).(pc, dpc, ddpc) = (8_{32}, 4_{32}, 0_{32}) \\ mc^0.p(q).(gpr, spr) &= h^0_\pi.p(q).(gpr, spr). \end{aligned}$$

For the invisible pipeline registers there is obviously nothing to show in cycle $t = 0$, after reset. Finally, for contents of the (specification) TLB and constructed TLB on all processors q we obtain trivially:

$$\begin{aligned} mc^0.p(q).tlb &\supseteq tlb_G(mmu_I^{q,0}) \cup tlb_G(mmu_E^{q,0}) = \emptyset \\ mc^0.p(q).tlb^\circ &\supseteq tlb_U(mmu_I^{q,0}) \cup tlb_U(mmu_E^{q,0}) = \emptyset. \end{aligned}$$

Note, from lemma 35 there is nothing to show for the walk registers of the MMUs, since the corresponding control automata reside in their idle states in cycle $t = 0$, after reset.

9.2.5 Stepping Inputs

Assume that processor q makes a step in cycle t and this step gets number n in the global ordering of ISA steps. Let's consider the TLB steps first and assume that step n is generated by mmu_Y. According to the definition (see Sect. 9.1.1), apart from the scheduling inputs

$$\begin{aligned} s(n).g &= mmu_Y \\ s(n).u &= q \in TS_Y(t) \end{aligned}$$

the stepping function also provides the oracle input. We define the oracle input for the latter TLB step as follows.

$$tstep_{Y\ G}(q,t) \ \rightarrow \ s(n).o = \begin{cases} (winit, upa_G(mmu_Y^{q,t}).pa, guest) & winit_G(mmu_Y^{q,t}) \\ (wext, mmu_Y^{q,t}.w_G, \perp) & \text{otherwise} \end{cases}$$

$$tstep_{Y\ U}(q,t) \ \rightarrow \ s(n).o = \begin{cases} (winit, upa_U(mmu_Y^{q,t}).pa, user) & winit_U(mmu_Y^{q,t}) \\ (wext, mmu_Y^{q,t}.w_U, mmu_Y^{q,t}.w_G) & \text{otherwise} \end{cases}$$

Now assume that step n is performed by the processor core.

$$\begin{aligned} s(n).t &= core \\ s(n).u &= q \in PS(t) \end{aligned}$$

Unlike the scheduling inputs, we assemble the oracle input for the processor core steps gradually, as the associated instruction advances through the pipeline to the lower/later stages. At the memory stage, when the processor core performs a step to execute that instruction, all fields of the stepping function must be completely specified. Further assume that processor q makes a step in cycle t to execute instruction

$$i = I(q, 6, t).$$

Instruction i progresses down through pipeline stages $k \in \{1, 5\}$ in cycles t_k such that

$$I(q, k, t_k) = i \wedge ue_k^{q,t_k}.$$

Note, due to possible misspeculations, there might be multiple cycles t_k in which instruction i progress in stage k. The oracle inputs for global step n are specified in those cycles t_k in which we in addition have

$$k > \mu(q,t_k).$$

The latter cycles are provably unique (see Sect. 8.4.5, equation 65). In cycles t_1 and t_5 we specify the first two components of the oracle input for step n.

$$s(n).o.w_I = \begin{cases} mmu_I^{q,t_1}.wout & mmu_I^{q,t_1}.treq \\ \perp & \text{otherwise} \end{cases}$$

$$s(n).o.w_E = \begin{cases} mmu_E^{q,t_5}.wout & mmu_E^{q,t_5}.treq \\ \perp & \text{otherwise} \end{cases}$$

Thus, the walk for translation of the instruction address is taken from the outputs of mmu_I once instruction i leaves the first translation stage for the last time. Analogously, if a memory operation is performed, the walk for translation of the effective address is taken from the outputs of mmu_E once instruction i leaves the second translation stage for the last time. Easy to see that the statements below follow from the definition above; we use these statements to argue about correctness for ghost walk registers in Sect. 9.6.3. Recall, the shorthands for the walk inputs were defined early in Sect. 9.1.2.

$$w_{I\ \sigma}^{q,I(q,1,t_1)} = \begin{cases} mmu_I^{q,t_1}.wout & mmu_I^{q,t_1}.treq \\ \perp & \text{otherwise} \end{cases} \tag{73}$$

$$w_{E\ \sigma}^{q,I(q,5,t_5)} = \begin{cases} mmu_E^{q,t_5}.wout & mmu_E^{q,t_5}.treq \\ \perp & \text{otherwise} \end{cases} \tag{74}$$

Finally, the vector of external events provided to processor q is sampled right before instruction i is executed in the memory stage. According to the assumption, the latter happens in cycle t.

$$s(n).o.eev = eev^{q,t}$$

From the definition above one can easily derive the following.

$$eev_{\sigma}^{q,I(q,6,t)} = eev^{q,t} \tag{75}$$

Note, since we assumed for the time being the absence of external interrupts, for all processors we have

$$eev^{q,t} = 00. \tag{76}$$

9.3 Developing Formalism

In this technical section we develop the machinery necessary to justify the induction hypothesis from Sect. 9.2.4. Thus, in Sect. 9.3.1 we establish important relationship between the scheduling functions (I) and instruction count (ic), whereas in Sect. 9.3.2 — between the steps of the processor ($pseq$) and steps of the ISA computation. Also, in Sect. 9.3.3 we elaborate on properties of the software speculation stage (Σ_{soft}).

9.3.1 Relating Instruction Count with Scheduling Functions

In this section we establish an intuitive relation between the number of instructions executed on the processor core q before global step $NS(t)$ and the index of instruction in the memory stage of processor q in cycle t.

Lemma 116.

$$ic(q,NS(t)) = I(q,6,t)$$

Proof of lemma 116. The proof is by an easy induction on the number of cycles t, and therefore we omit it. For details we refer to the proof of lemma 9.15 of [KMP14]. $\square$

Clearly, the instruction count for processor q changes at most once in the interval

$$[NS(t) : NS(t+1)]$$

since each processor can be stepped at most once in cycle t. In the following lemma we identify the steps at which the latter changes occur.

Lemma 117. *Assume that processor q performs a core step in cycle t, i.e.,*

$$s(NS(t)+y).(t,u) = (core,q).$$

Then the following holds for $n \leq ns(t)$:

$$ic(q,NS(t)+n) = \begin{cases} ic(q,NS(t)) & n \leq y \\ ic(q,NS(t+1)) & otherwise. \end{cases}$$

Proof of lemma 117. First, by induction on n we show the following.

$$n \leq y \ \rightarrow \ ic(q,NS(t)+n) = ic(q,NS(t)) \tag{77}$$

The base case ($n=0$) holds trivially. For the induction step from n to $n+1 \leq y$ we argue as follows. According to the definition of the stepping function (equation 67) we know

$$n < y \ \rightarrow \ s(NS(t)+n).(t,u) \neq (core,q)$$

and therefore we have

$$\begin{aligned} ic(q,NS(t)+n+1) &= ic(q,NS(t)+n) && \text{(definition)} \\ &= ic(q,NS(t)) && \text{(induction hypothesis).} \end{aligned}$$

Next, directly from the definition, for $n = y+1$ we obtain

$$\begin{aligned} ic(q,NS(t)+n) &= ic(q,NS(t)+y)+1 && \text{(definition)} \\ &= ic(q,NS(t))+1 && \text{(equation 77).} \end{aligned} \tag{78}$$

Finally, by induction on n we show the following.

$$n > y+1 \ \rightarrow \ ic(q,NS(t)+n) = ic(q,NS(t))+1 \tag{79}$$

Again using equation 67 we derive

$$n > y \ \rightarrow \ s(NS(t)+n).(t,u) \neq (core,q).$$

For the base case ($n = y+2$) we argue as follows.

$$\begin{aligned} ic(q,NS(t)+y+2) &= ic(q,NS(t)+y+1) && \text{(definition)} \\ &= ic(q,NS(t))+1 && \text{(equation 78)} \end{aligned}$$

For the induction step from n to $n+1 \leq ns(t)$ we argue as follows.

$$\begin{aligned} ic(q,NS(t)+n+1) &= ic(q,NS(t)+n) && \text{(definition)} \\ &= ic(q,NS(t))+1 && \text{(induction hypothesis)} \end{aligned}$$

In order to obtain the claim we simply rewrite Eqs. 78 and 79 using

$$\begin{aligned} ic(q,NS(t+1)) &= ic(q,NS(t)+ns(t)) && \text{(definition)} \\ &= ic(q,NS(t))+1 && \text{(equation 79).} \end{aligned}$$ $\square$

For convenience we introduce one more technical result, which follows directly from lemmas 117 and 116.

Lemma 118. *Assume that processor q performs a core step in cycle t, i.e.,*

$$s(n).(t,u) = (core,q) \wedge n \in CS(t).$$

Then

$$ic(q,n) = I(q,6,t).$$

9.3.2 Relating Global with Processor Local Steps

In order to show correctness for the instruction cache output, we require the following technical result.

Lemma 119. *Let $i = I(q,2,t)$. Then the following holds.*

$$[NS(t) : NS(t+1)-1] \subseteq [pseq(q,i-5)+1 : pseq(q,i+1)-1]$$

Proof of lemma 119. First, from monotonicity of function *pseq* we obtain

$$\begin{aligned} NS(t) \le NS(t+1) &\le pseq(q,ic(q,NS(t+1))) && \text{(lemma 111)} \\ &= pseq(q,I(q,6,t+1)) && \text{(lemma 116)} \\ &\le pseq(q,I(q,6,t)+1) && \text{(lemma 80)} \\ &\le pseq(q,i+1) && \text{(lemma 81).} \end{aligned}$$

Then, again from monotonicity of *pseq* we derive

$$\begin{aligned} NS(t+1) \ge NS(t) &> pseq(q,ic(q,NS(t)) - 1) && \text{(lemma 112)} \\ &= pseq(q,I(q,6,t)-1) && \text{(lemma 116)} \\ &\ge pseq(q,i-5) && \text{(lemma 81).} \end{aligned}$$

□

9.3.3 Properties of Σ

In the scope of this section we derive properties of the software speculation stage (Σ_{soft}). The corresponding properties of other speculation stages will be shown later, in Sect. 9.4.4, after we weaken the assumptions of the lemma below (equation 88).

Lemma 120. *Assume that step n (in the global ordering of ISA steps) is a core step of processor q performed in cycle t.*

$$s(n).(t,u) = (core,q) \wedge n \in CS(t) \qquad \text{i)}$$

Assume that the ISA computation is guarded until step n and after the fetch in step n.

$$\Gamma^n \wedge \Phi_I(mc^n.p(q), s(n).o) \qquad \text{ii)}$$

Then

$$\Sigma_{soft}(q,t) < 5.$$

Proof of lemma 120. From the assumptions we know that the instruction executed in step *n* (by processor *q*) does not violate the software conditions for instruction fetch. Formally, we argue

$$\begin{aligned} \Gamma^n \wedge \Phi_I(mc^n.p(q), s(n).o) &\to SC_{code}(q,ic(q,n)) && \text{(equation 71)} \\ &\to SC_{code}(q,I(q,6,t)) && \text{(lemma 118)} \\ &\to SC_{code}(q,I(q,5,t)-1) && \text{(lemma 79)} \end{aligned}$$

and the claim follows. □

In the presence of a misspeculation, the value of the software speculation stage (Σ_{soft}) in the next cycle is given in the following lemma. Note, for convenience processor indices are omitted below in this section.

Lemma 121. *Assume $\Sigma_{soft}(t) = \sigma$ such that $\sigma > 0$.*

$$\overline{ue^t_{\sigma+1}} \to \Sigma_{soft}(t+1) = \begin{cases} 0 & rbr^t_{\sigma+1} \\ \sigma & otherwise \end{cases} \qquad (1)$$

$$\sigma < 5 \wedge ue^t_{\sigma+1} \to \Sigma_{soft}(t+1) = \sigma + 1 \qquad (2)$$

Proof of lemma 121. Assume

$$\begin{aligned}\Sigma_{soft}(t) &= \sigma_0 > 0\\ \Sigma_{soft}(t+1) &= \sigma_1.\end{aligned}$$

By definition of Σ_{soft} we have the following.

$$rfull^t_{\sigma_0} \wedge /SC_{code}(I(\sigma_0,t)-1) \tag{80}$$

$$\nexists k > \sigma_0 : \; rfull^t_k \wedge /SC_{code}(I(k,t)-1) \tag{81}$$

First we show that the following holds.

$$\nexists k > \sigma_0 + 1 : \; rfull^{t+1}_k \wedge /SC_{code}(I(k,t+1)-1) \tag{82}$$

Proof of equation 82. By contradiction. Assume there is such stage

$$k > \sigma_0 + 1$$

and it was updated in cycle t. From the definitions of the update enable signal (equation 35) and the scheduling functions we resp. have

$$ue^t_k \;\rightarrow\; rfull^t_{k-1}$$

and (since $k > 1$)

$$ue^t_k \;\rightarrow\; I(k,t+1) = I(k-1,t).$$

From the assumption and arguments above we derive

$$rfull^t_{k-1} \wedge /SC_{code}(I(k-1,t)-1)$$

which leads to a contradiction:

$$\Sigma_{soft}(t) \geq k-1 > \sigma_0.$$

Otherwise, if stage k was not updated in cycle t, from the definitions of the full and the rollback-pending bits we resp. have

$$full^{t+1}_k \wedge /ue^t_k \;\rightarrow\; stall^t_{k+1} \wedge /rollback^t_k$$

and (using equation 36)

$$/rbp^{t+1}_k \wedge /rollback^t_k \;\rightarrow\; /rbp^t_k \wedge /rbr^t_{k+1}.$$

From the definitions of the stall and update enable signals we resp. have

$$stall^t_{k+1} \;\rightarrow\; full^t_k$$

and

$$stall^t_{k+1} \;\rightarrow\; /ue^t_{k+1}.$$

Thus, from the assumption and arguments above we conclude

$$rfull^t_k \wedge /SC_{code}(I(k,t+1)-1)$$

where for the instruction index we derive:

$$\begin{aligned}I(k,t+1)-1 &= I(k+1,t+1) && \text{(lemma 79)}\\ &= I(k+1,t) && \text{(definition)}\\ &= I(k,t)-1 && \text{(lemma 79).}\end{aligned}$$

The latter gives a contradiction:

$$\Sigma_{soft}(t) \geq k > \sigma_0 + 1.$$

□

Next we split cases on the values of the rollback request and update enable signals of stage $\sigma_0 + 1$ in cycle t:

- $ue^t_{\sigma_0+1} \wedge \sigma_0 < 5$. In this case we are to show
$$\sigma_1 = \sigma_0 + 1.$$
From the first part of lemma 75 and definition of the scheduling functions we resp. have
$$ue^t_{\sigma_0+1} \rightarrow rfull^{t+1}_{\sigma_0+1}$$
and (since $\sigma_0 > 0$)
$$ue^t_{\sigma_0+1} \rightarrow I(\sigma_0+1, t+1) = I(\sigma_0, t).$$
From equation 80 and the arguments above we conclude
$$rfull^{t+1}_{\sigma_0+1} \wedge /SC_{code}(I(\sigma_0+1, t+1) - 1)$$
which by definition implies
$$\Sigma_{soft}(t+1) \geq \sigma_0 + 1.$$
- $\overline{ue^t_{\sigma_0+1}} \wedge \overline{rbr^t_{\sigma_0+1}}$. In this case we have to show
$$\sigma_1 = \sigma_0.$$
First, using part (5) of lemma 75 we derive
$$rfull^t_{\sigma_0} \wedge /ue^t_{\sigma_0+1} \wedge /rbr^t_{\sigma_0+1} \rightarrow rfull^{t+1}_{\sigma_0}.$$
Thus, from equation 80 and the arguments above we conclude
$$rfull^{t+1}_{\sigma_0} \wedge /SC_{code}(I(\sigma_0, t) - 1)$$
where for the instruction index we derive:
$$\begin{aligned} I(\sigma_0, t) - 1 &= I(\sigma_0+1, t) && \text{(lemma 79)} \\ &= I(\sigma_0+1, t+1) && \text{(definition)} \\ &= I(\sigma_0, t+1) - 1 && \text{(lemma 79).} \end{aligned}$$
The latter by definition implies
$$\Sigma_{soft}(t+1) \geq \sigma_0.$$
Next we proceed to show
$$\Sigma_{soft}(t+1) \neq \sigma_0 + 1.$$
We assume that the stage below σ_0 has a real full bit in cycle $t+1$, since otherwise the claim follows. From the definitions of the full and the rollback-pending bits we resp. have
$$full^{t+1}_{\sigma_0+1} \wedge /ue^t_{\sigma_0+1} \rightarrow stall^t_{\sigma_0+2} \wedge /rollback^t_{\sigma_0+1}$$
and (using equation 36)
$$/rbp^{t+1}_{\sigma_0+1} \wedge /rollback^t_{\sigma_0+1} \rightarrow /rbp^t_{\sigma_0+1}.$$
Finally, from the definitions of the stall signal and the scheduling functions we resp. have
$$stall^t_{\sigma_0+2} \rightarrow full^t_{\sigma_0+1}$$
and
$$/ue^t_{\sigma_0+1} \wedge /rbr^t_{\sigma_0+1} \rightarrow I(\sigma_0+1, t+1) = I(\sigma_0+1, t).$$
From equation 81 and the arguments above we conclude
$$SC_{code}(I(\sigma_0+1, t+1) - 1)$$
which by definition implies
$$\Sigma_{soft}(t+1) \neq \sigma_0 + 1.$$

- $\overline{ue^t_{\sigma_0+1}} \wedge rbr^t_{\sigma_0+1}$. Here we proceed to show

  $$\sigma_1 = 0.$$

  Using part (2) of lemma 75 we conclude

  $$\nexists k < \sigma_0 + 1 : \ rfull_k^{t+1}$$

  which by definition implies

  $$\Sigma_{soft}(t+1) \in \{0\} \cup [\sigma_0 + 1 : 5].$$

  Finally, for $r = R(t)$ such that

  $$\sigma_0 < r \leq 5$$

  from the definition of the *misspec* signals we conclude

  $$rbr^t_r \ \rightarrow \ rfull^t_r.$$

  Therefore, from the definition of the scheduling functions we have

  $$/ue^t_{\sigma_0+1} \wedge rbr^t_{\sigma_0+1} \ \rightarrow \ I(\sigma_0+1, t+1) = I(r,t)$$

  and from equation 81 we conclude

  $$SC_{code}(I(\sigma_0+1, t+1) - 1).$$

  The latter by definition implies

  $$\Sigma_{soft}(t+1) \neq \sigma_0 + 1.$$

Equation 82 gives

$$\Sigma_{soft}(t+1) \leq \sigma_0 + 1$$

which completes the proof in all cases above. □

For convenience we generalize the result above in the following lemma.

Lemma 122. *Assume* $\Sigma_{soft}(t) = \sigma$.

$$\Sigma_{soft}(t+1) \leq \sigma + ue^t_{\sigma+1}$$

Proof of lemma 122. For $\Sigma_{soft}(t) > 0$ the claim follows directly from lemma 121. For $\Sigma_{soft}(t) = 0$ we proceed to show the following.

$$\Sigma_{soft}(t+1) \leq ue^t_1$$

Repeating the arguments presented in the proof of equation 82 we derive

$$\nexists k > 1 : \ rfull_k^{t+1} \wedge /SC_{code}(I(k, t+1) - 1)$$

which by definition implies

$$\Sigma_{soft}(t+1) \leq 1.$$

Therefore, if stage 1 is updated in cycle t (ue^t_1), the claim follows. Otherwise, we argue by contradiction; assume $/ue^t_1$ and

$$\Sigma_{soft}(t+1) = 1.$$

From the definition we have

$$rfull_1^{t+1} \wedge /SC_{code}(I(1, t+1) - 1).$$

Using parts (2) and (4) of lemma 75 we resp. conclude

$$/rbr_2^t \wedge rfull_1^t.$$

From the definition of the *misspec* signals we further derive

$$/rbr_1^t$$

and therefore for the scheduling function in stage 1 by definition we obtain

$$I(1,t+1) = I(1,t).$$

From the arguments above we clearly have

$$\Sigma_{soft}(t) = 1$$

which is a contradiction. □

9.4 Verifying Guard Conditions

Analogous to Sect. 7.5, we are to show that the guard conditions are respected in all ISA steps until $NS(t+1)$, i.e.,

$$\forall n < NS(t+1) : \ \Gamma(mc^n, s(n)).$$

Directly from the induction hypothesis we obtain:

$$\forall n < NS(t) : \ \Gamma(mc^n, s(n)).$$

For the remaining steps we proceed to show

$$\forall n \in [NS(t) : NS(t) + \ell - 1] : \ \Gamma(mc^n, s(n))$$

by induction on ℓ, where ℓ denotes the length of sub-interval

$$[NS(t) : NS(t) + \ell - 1] \subseteq [NS(t) : NS(t+1) - 1].$$

The base case holds trivially since for $\ell = 0$ there is nothing to show. For the induction step from $\ell - 1$ to $\ell \leq ns(t)$ we split cases on whether step

$$n = NS(t) + \ell$$

is a TLB step (see Sect. 9.4.1) or a processor core step (see Sect. 9.4.2).

Before we proceed to the induction step, we reformulate the result (from Sect. 7.5) for the implementation registers of the nested MMUs. Thus, in case a walk extension step is performed by mmu_Y of processor q in cycle t, the implementation register $mmu_Y.w_X$ contains valid walks on both, guest and user TLB steps.

Lemma 123. *Assume $q \in TA_Y(t)$.*

$$tstep_{Y\ G}(q,t) \ \rightarrow \ valid(mc^{NS(t)}.p(q).tlb, mmu_Y^{q,t}.w_G) \qquad (1)$$

$$tstep_{Y\ U}(q,t) \ \rightarrow \ valid(mc^{NS(t)}.p(q).tlb, mmu_Y^{q,t}.w_U, mmu_Y^{q,t}.w_G) \qquad (2)$$

Note, we can use the proof of lemma 72 literally, since the arguments involved rely entirely on the internal construction of the nested MMU, namely on the construction of its control mechanisms.

9.4.1 TLB Steps

Assume that step n is generated by mmu_Y on processor q.

$$s(n).(g,u) = (mmu_Y, q)$$

Below we consider only those cases in which mmu_Y of processor q is not invalidated in cycle t ($/inval(mmu_Y^{q,t})$), since otherwise we immediately derive a contradiction.

$$\begin{aligned} inval(mmu_Y^{q,t}) &\rightarrow /mmu_Y^{q,t}.treq && \text{(definition)} \\ &\rightarrow /tstep_Y(q,t) && \text{(definition)} \end{aligned}$$

In case processor q performs a core step in cycle t and this step gets number $m < n$ in the global ordering of ISA steps, we apply lemma 120 to argue that

$$\Sigma_{soft}(q,t) < 5$$

which allows us to use the induction hypothesis for non-local invisible registers in stage 5 to argue that neither $invlpg$ nor $flusht$ are executed by processor q in step m.

$$s(m).(t,u) = (core,q) \wedge m \in [NS(t) : n-1] \;\rightarrow\; /(invlpg_\sigma \vee flusht_\sigma)^{q,ic(q,m)} \tag{83}$$

Similarly we argue that execution mode on processor q does not decrease in cycle t

$$\begin{aligned} mode(q,t) > mode(q,t+1) &\rightarrow jisr(q,t) && \text{(lemma 78.1)} \\ &\rightarrow /mmu_Y^{q,t}.treq && \text{(definition)} \\ &\rightarrow /tstep_Y(q,t) && \text{(definition)} \end{aligned}$$

and thus consider the following three cases that remain possible.

i) Remaining at the level of guest.

$$guest(q,t) \wedge guest(q,t+1)$$

As above we argue that $jisr$ or $eret$ are not executed by processor q in step m.

$$s(m).(t,u) = (core,q) \wedge m \in [NS(t) : n-1] \;\rightarrow\; /(jisr_\sigma \vee eret_\sigma)^{q,ic(q,m)} \tag{84}$$

From the induction hypothesis for visible register in the memory stage we derive

$$guest(mc^{NS(t)}.p(q))$$

which together with the result above (equation 84) and the fact that any "moves" to the mode registers are illegal at the level of guest (see Sect. 3.3.6) gives

$$guest(mc^n.p(q)).$$

From interconnect and construction of the nested MMU we derive:

$$\begin{aligned} guest(q,t) &\rightarrow mmu_Y^{q,t}.upa \notin A_U && \text{(interconnect)} \\ &\rightarrow /treq_U(mmu_Y^{q,t}) && \text{(definition)} \\ &\rightarrow /tstep_{Y\ U}(q,t) && \text{(definition).} \end{aligned}$$

Thus, in case mmu_Y performs initialization of a guest walk ($winit_G(mmu_Y^{q,t})$), for the stepping inputs by definition we have

$$s(n).o.(t,l) = (winit, guest).$$

From the arguments above we easily conclude

$$T_G(mc^n.p(q), s(n).o).$$

In case mmu_Y performs extension of a guest walk ($wext_G(mmu_Y^{q,t})$), for the stepping inputs by definition we have

$$s(n).o = (wext, mmu_Y^{q,t}.w_G, \perp).$$

From the first part of lemma 123 we have

$$valid(mc^{NS(t)}.p(q).tlb, mmu_Y^{q,t}.w_G)$$

which together with the result above (equation 83) gives

$$valid(mc^n.p(q).tlb, mmu_Y^{q,t}.w_G).$$

From the arguments above we again conclude

$$T_G(mc^n.p(q), s(n).o).$$

ii) Remaining at the level of user.

$$user(q,t) \wedge user(q,t+1)$$

Again, we argue that *jisr* or *eret* are not executed by processor q in step m.

$$s(m).(t,u) = (core, q) \wedge m \in [NS(t) : n-1] \;\rightarrow\; /(jisr_\sigma \vee eret_\sigma)^{q,ic(q,m)} \qquad (85)$$

From the induction hypothesis for visible register in the memory stage we derive

$$user(mc^{NS(t)}.p(q))$$

which together with the result above (equation 85) and the fact that any "moves" to the mode registers are illegal at the level of user (see Sect. 3.3.6) gives

$$user(mc^n.p(q)).$$

In case mmu_Y performs initialization of a guest walk ($winit_G(mmu_Y^{q,t})$) or extension of a guest walk ($wext_G(mmu_Y^{q,t})$) the arguments are identical to those presented in case i). Further, in case mmu_Y performs initialization of a user walk ($winit_U(mmu_Y^{q,t})$), for the stepping inputs by definition we have

$$s(n).o.(t,l) = (winit, user).$$

From the arguments above we easily conclude

$$T_U(mc^n.p(q), s(n).o).$$

Finally, in case mmu_Y performs extension of a user walk ($wext_U(mmu_Y^{q,t})$), for the stepping inputs by definition we have

$$s(n).o = (wext, mmu_Y^{q,t}.w_U, mmu_Y^{q,t}.w_G).$$

From the second part of lemma 123 we have

$$valid(mc^{NS(t)}.p(q).tlb, mmu_Y^{q,t}.w_U, mmu_Y^{q,t}.w_G)$$

which together with the result above (equation 83) gives

$$valid(mc^n.p(q).tlb, mmu_Y^{q,t}.w_U, mmu_Y^{q,t}.w_U).$$

From the arguments above we again conclude

$$T_U(mc^n.p(q), s(n).o).$$

iii) Execution of *eret* at the level of guest.

$$guest(q,t) \wedge user(q,t+1)$$

Finally, analogous to above we argue that processor q executes *eret* in step m.

$$s(m).(t,u) = (core,q) \wedge m \in [NS(t) : n-1] \;\rightarrow\; eret_{\sigma}^{q,ic(q,m)} \tag{86}$$

From the induction hypothesis for visible register in the memory stage we derive

$$guest(mc^{NS(t)}.p(q))$$

which together with the result above (equation 86) gives

$$guest(mc^{n}.p(q)) \vee user(mc^{n}.p(q))$$

depending on whether a core step or a TLB step of processor q is performed first in cycle t. The remaining proof lines literally repeat the arguments presented in case i).

9.4.2 Processor Core Steps

Assume that step n is performed by processor q.

$$s(n).(t,u) = (core,q)$$

Using the induction hypothesis, we can rewrite the stepping given above in Sect. 9.2.5 in terms of the registers in stage 5 of the ghost walk pipeline as follows.

Lemma 124. *Assume that processor q performs a core step in cycle t, i.e.,*

$$s(n).(t,u) = (core,q) \wedge n \in CS(t).$$

Then the following holds:

$$used(w_I,n) \;\rightarrow\; s(n).o.w_I \,=\, w_I.5^{q,t} \tag{1}$$

$$used(w_E,n) \;\rightarrow\; s(n).o.w_E \,=\, w_E.5^{q,t} \tag{2}$$

Proof of lemma 124.1. For the first walk passed as the machine's input we derive:

$$\begin{aligned} s(n).o.w_I &= w_{I\ \sigma}^{q,ic(q,n)} && \text{(lemma 110)} \\ &= w_{I\ \sigma}^{q,I(q,6,t)} && \text{(lemma 118)} \\ &= w_{I\ \sigma}^{q,I(q,5,t)-1} && \text{(lemma 79)} \\ &= w_I.5^{q,t} && \text{(IH).} \end{aligned}$$

□

That the walks coming from the MMUs — from the ghost pipeline — satisfy the guard conditions of the specification was reflected in the induction hypothesis above. In case the first walk passed to the specification machine is used (for translation of the instruction address)

$$used(w_I,n)$$

it is contained in the corresponding TLB (invariant 10)

$$w_I.5^{q,t} \in \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases}$$

and matches the corresponding translation request (invariant 11).

$$match(trq_{I\ \sigma}^{q,I(q,5,t)-1}, w_I.5^{q,t}) \leftrightarrow match(trq_{I\ \sigma}^{q,I(q,6,t)}, w_I.5^{q,t}) \qquad \text{(lemma 79)}$$
$$\leftrightarrow match(trq_{I\ \sigma}^{q,ic(q,n)}, w_I.5^{q,t}) \qquad \text{(lemma 118)}$$

Since there are no other core steps (except for step n) of processor q in cycle t

$$m \in [NS(t) : n-1] \ \rightarrow \ s(m).(t,u) \neq (core, q)$$

the TLB on processor q cannot decrease in size (before step n).

$$w_I.5^{q,t} \in \begin{cases} mc^n.p(q).tlb^\circ & user(mc^n.p(q)) \\ mc^n.p(q).tlb & guest(mc^n.p(q)) \end{cases}$$

Applying lemma 124 (first part) we conclude

$$\Phi_I(mc^n.p(q), s(n).o).$$

Combining the induction hypotheses with the result above, we derive that the ISA computation is guarded until step n and after the fetch in step n.

$$\Gamma^{NS(t)} \wedge \forall m \in [NS(t) : n-1] : \ \Gamma(mc^m, s(m)) \wedge \Phi_I(mc^n.p(q), s(n).o) \qquad (87)$$

The latter allows us to apply the software condition, and proceed to the proof of the second part of lemma 124.

Proof of lemma 124.2. For the second walk passed to the specification machine we argue repeating the proof of part (1) (of lemma 124) as follows:

$$\begin{aligned} s(n).o.w_E &= w_{E\ \sigma}^{q,ic(q,n)} && \text{(lemma 110)} \\ &= w_{E\ \sigma}^{q,I(q,6,t)} && \text{(lemma 118)} \\ &= w_{E\ \sigma}^{q,I(q,5,t)-1} && \text{(lemma 79)} \\ &= w_E.5^{q,t} && \text{(equation 87; lemma 120; IH).} \end{aligned}$$ □

In case the second walk passed to the specification machine is used (for translation of the effective address)

$$used(w_E, n)$$

using lemma 120 we argue that the latter walk is contained in the corresponding TLB (invariant 10)

$$w_E.5^{q,t} \in \begin{cases} mc^{NS(t)}.p(q).tlb^\circ & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases}$$

and matches the corresponding translation request (invariant 11).

$$match(trq_{E\ \sigma}^{q,I(q,5,t)-1}, w_E.5^{q,t})$$

Repeating the arguments above, we apply lemma 124 (second part) and conclude

$$\Phi_E(mc^n.p(q), s(n).o).$$

Combining equation 87 with the result above, we derive that the ISA computation is guarded until step $n+1$

$$\Gamma^{NS(t)} \wedge \forall m \in [NS(t) : n] : \ \Gamma(mc^m, s(m))$$

which completes the induction step.

9.4.3 Processor Control Signals

In Sect. 9.4 we showed that the ISA computation is guarded until step $NS(t+1)$. The latter allows us to rewrite lemma 120 simply as follows.

$$ue_6^t \rightarrow \Sigma_{soft}(t) < 5 \tag{88}$$

Lemma 125. *Assume processor q performs a core step in cycle t ($cstep(q,t)$).*

1. $$mca(6)^{q,t} = mca(6)_\sigma^{q,I(q,6,t)}$$

2. $$jisr(q,t) = jisr_\sigma^{q,I(q,6,t)}$$

Proof of lemma 125.1. First we show correctness for the event signals collected in the memory stage.

$$ev(6)^{q,t} = ev(6)_\sigma^{q,I(q,6,t)} \tag{89}$$

Proof of equation 89.

$$\begin{aligned} ev(6)^{q,t} &= 0_9 \circ eev^{q,t} && \text{(definition)} \\ &= 0_9 \circ s(n).o.eev && \text{(stepping)} \\ &= 0_9 \circ eev_\sigma^{q,ic(q,n)} && \text{(lemma 110)} \\ &= 0_9 \circ eev_\sigma^{q,I(q,6,t)} && \text{(lemma 118)} \\ &= ev(6)_\sigma^{q,I(q,6,t)} && \text{(definition)} \end{aligned}$$ □

For the masked cause signal of the memory stage we argue as follows.

$$\begin{aligned} mca(6)^t &= (ev(6)^t \vee ca.5_\pi^t) \wedge imask^t && \text{(definition)} \\ &= (ev(6)^t \vee ca.5_\pi^t) \wedge (1^9 \circ sr_\pi^t[1] \circ 1) && \text{(definition)} \\ &= (ev(6)_\sigma^{I(6,t)} \vee ca.5_\sigma^{I(6,t)}) \wedge (1^9 \circ sr_\sigma^{I(6,t)}[1] \circ 1) && \text{(equations 88–89; IH)} \\ &= (ev(6)_\sigma^{I(6,t)} \vee ca.5_\sigma^{I(6,t)}) \wedge imask_\sigma^{I(6,t)} && \text{(definition)} \\ &= mca(6)_\sigma^{I(6,t)} && \text{(definition)} \end{aligned}$$ □

Proof of lemma 125.2. For the *jisr* signal we argue using the result above.

$$\begin{aligned} jisr(t) &\leftrightarrow ue_6^t \wedge jisr.5_\pi^t && \text{(definition)} \\ &\leftrightarrow ue_6^t \wedge (mca(6)^t \neq 0_{11}) && \text{(invariant 8)} \\ &\leftrightarrow mca(6)_\sigma^{I(6,t)} \neq 0_{11} && \text{(lemma 125.1)} \\ &\leftrightarrow jisr_\sigma^{I(6,t)} && \text{(definition)} \end{aligned}$$ □

Later on, when we consider interrupts, we require the latter results to argue about correctness for the special purpose registers.

9.4.4 Properties of Σ Revisited

We continue developing the arguments presented in Sect. 9.3.3.

Properties of Σ

For speculation stage Σ we show the results analogous to the corresponding results about the software speculation stage (Σ_{soft}). First we argue that speculation stage Σ does not "overflow", i.e., takes values only in range $[0:5]$.

Lemma 126. *Assume $\Sigma(t) = 5$.*

$$ue_6^t \rightarrow rbr_5^t$$

Proof of lemma 126. By contradiction. From the definition of signal $misspec_5$ we have

$$jisr(t) = 0.$$

The contradiction follows from part (2) of lemma 125.

$$\Sigma(t) < 5$$

□

The value of speculation stage Σ in the next cycle is given in the following lemma.

Lemma 127. *Assume $\Sigma(t) > 0$ and $\Sigma(t) \geq R(t)$.*

$$\overline{ue_{\Sigma(t)+1}^t} \rightarrow \Sigma(t+1) = \Sigma(t) \tag{1}$$

$$ue_{\Sigma(t)+1}^t \rightarrow \Sigma(t+1) = \begin{cases} 0 & rbr_{\Sigma(t)}^t \\ \Sigma(t)+1 & otherwise \end{cases} \tag{2}$$

Proof of lemma 127.1. By contradiction. Assume

$$\Sigma(t) = \sigma_0 > 0$$
$$\Sigma(t+1) = \sigma_1 \neq \sigma_0.$$

By definition of Σ we have the following.

$$rfull_{\sigma_0}^t \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_0,t)-1} \tag{90}$$

$$rfull_{\sigma_1}^{t+1} \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_1,t+1)-1} \tag{91}$$

In case

$$\sigma_0 < \sigma_1$$

we split cases on $ue_{\sigma_1}^t$. In case stage σ_1 is updated in cycle t, from the definitions of the update enable signal (equation 35) and the scheduling functions we resp. have

$$ue_{\sigma_1}^t \rightarrow rfull_{\sigma_1-1}^t$$

and (since $\sigma_1 > 1$)

$$ue_{\sigma_1}^t \rightarrow I(\sigma_1, t+1) = I(\sigma_1 - 1, t).$$

From equation 91 and the arguments above we conclude

$$rfull_{\sigma_1-1}^t \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_1-1,t)-1}$$

which by definition implies

$$\Sigma(t) \geq \sigma_1 - 1.$$

From the assumption ($\sigma_0 < \sigma_1$) we also derive

$$\Sigma(t) \leq \sigma_1 - 1.$$

The contradiction follows, since by assumption ($ue_{\sigma_1}^t$) we obtain

$$ue_{\Sigma(t)+1}^t.$$

Otherwise, if stage σ_1 is not updated in cycle t, from the definitions of the full and the rollback-pending bits we resp. have

$$full^{t+1}_{\sigma_1} \wedge /ue^t_{\sigma_1} \ \rightarrow \ stall^t_{\sigma_1+1} \wedge /rollback^t_{\sigma_1}$$

and (using equation 36)

$$/rbp^{t+1}_{\sigma_1} \wedge /rollback^t_{\sigma_1} \ \rightarrow \ /rbp^t_{\sigma_1}.$$

From the definitions of the stall signal and the scheduling functions we resp. have

$$stall^t_{\sigma_1+1} \ \rightarrow \ full^t_{\sigma_1}$$

and (since $\sigma_1 > R(t)$)

$$/ue^t_{\sigma_1} \ \rightarrow \ I(\sigma_1, t+1) = I(\sigma_1, t).$$

Thus, from equation 91 and the arguments above we conclude

$$rfull^t_{\sigma_1} \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_1,t)-1}$$

which by definition implies

$$\Sigma(t) \geq \sigma_1.$$

The latter is a contradiction, since as an assumption ($\sigma_0 < \sigma_1$) we have

$$\Sigma(t) < \sigma_1.$$

In case

$$\sigma_1 < \sigma_0$$

using part (5) of lemma 75 we derive

$$rfull^t_{\sigma_0} \wedge /ue^t_{\sigma_0+1} \wedge /rbr^t_{\sigma_0+1} \ \rightarrow \ rfull^{t+1}_{\sigma_0}.$$

Since $\sigma_0 \geq R(t)$, from equation 90 and the arguments above we conclude

$$rfull^{t+1}_{\sigma_0} \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_0,t)-1}$$

where for the instruction index we derive:

$$\begin{aligned} I(\sigma_0,t) - 1 &= I(\sigma_0+1,t) && \text{(lemma 79)} \\ &= I(\sigma_0+1,t+1) && \text{(definition)} \\ &= I(\sigma_0,t+1) - 1 && \text{(lemma 79).} \end{aligned}$$

The latter implies

$$\Sigma(t+1) \geq \sigma_0$$

which is a contradiction, since as an assumption ($\sigma_1 < \sigma_0$) we have

$$\Sigma(t+1) < \sigma_0. \qquad \square$$

Proof of lemma 127.2. Assume

$$\begin{aligned} \Sigma(t) &= \sigma_0 > 0 \\ \Sigma(t+1) &= \sigma_1. \end{aligned}$$

By definition of Σ we have the following.

$$rfull^t_{\sigma_0} \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_0,t)-1} \tag{92}$$

First we show that the following holds.

$$\nexists k > \sigma_0 + 1 : \ rfull^{t+1}_k \wedge (jisr_\sigma \vee eret_\sigma)^{I(k,t+1)-1} \tag{93}$$

Proof of equation 93. By contradiction. Assume there is such stage

$$k > \sigma_0 + 1$$

and it was updated in cycle t. From the definitions of the update enable signal (equation 35) and the scheduling functions we resp. have

$$ue_k^t \rightarrow rfull_{k-1}^t$$

and (since $k > 1$)

$$ue_k^t \rightarrow I(k,t+1) = I(k-1,t).$$

From the assumption and arguments above we derive

$$rfull_{k-1}^t \wedge (jisr_\sigma \vee eret_\sigma)^{I(k-1,t)-1}$$

which leads to a contradiction:

$$\Sigma(t) \geq k-1 > \sigma_0.$$

Otherwise, if stage k was not updated in cycle t, from the definitions of the full and the rollback-pending bits we resp. have

$$full_k^{t+1} \wedge /ue_k^t \rightarrow stall_{k+1}^t \wedge /rollback_k^t$$

and (using equation 36)

$$/rbp_k^{t+1} \wedge /rollback_k^t \rightarrow /rbp_k^t.$$

From the definitions of the stall signal and the scheduling functions we resp. have

$$stall_{k+1}^t \rightarrow full_k^t$$

and (since $k > R(t)$)

$$/ue_k^t \rightarrow I(k,t+1) = I(k,t).$$

Thus, from the assumption and arguments above we conclude

$$rfull_k^t \wedge (jisr_\sigma \vee eret_\sigma)^{I(k,t)-1}$$

which gives a contradiction:

$$\Sigma(t) \geq k > \sigma_0 + 1.$$ □

Further we split cases on the value of σ_0:

- $\sigma_0 \in \{1,3,4\}$. Since $\sigma_0 \geq R(t)$, the rollback request signal in stage σ_0 is inactive in cycle t.

  $$rbr_{\sigma_0}^t = 0$$

  Therefore, we are to show

  $$\sigma_1 = \sigma_0 + 1.$$

  From the first part of lemma 75 and definition of the scheduling functions we resp. have

  $$ue_{\sigma_0+1}^t \rightarrow rfull_{\sigma_0+1}^{t+1}$$

  and (since $\sigma_0 > 0$)

  $$ue_{\sigma_0+1}^t \rightarrow I(\sigma_0+1,t+1) = I(\sigma_0,t).$$

  Thus, from equation 92 and the arguments above we conclude

  $$rfull_{\sigma_0+1}^{t+1} \wedge (jisr_\sigma \vee eret_\sigma)^{I(\sigma_0+1,t+1)-1}$$

  which by definition implies

  $$\Sigma(t+1) \geq \sigma_0 + 1.$$

  Equation 93 gives

  $$\Sigma(t+1) \leq \sigma_0 + 1$$

  and the claim follows.

- $\sigma_0 = 2$. By definition (of Σ) we have

$$rfull_2^t \wedge (jisr_\sigma \vee eret_\sigma)^{I(2,t)-1}.$$

We split cases on the value of the rollback request signal of stage 2 in cycle t. (Note, from the assumptions we know $\sigma_0 \geq R(t)$.)

$$\begin{aligned} rbr_2^t &\leftrightarrow rfull_2^t \wedge iret_2^t && \text{(interconnect)} \\ &\leftrightarrow eret(3)^t \wedge mca(3)^t[>\texttt{il}] && \text{(definition)} \\ &\leftrightarrow eret_\sigma^{I(2,t)-1} \wedge mca_\sigma^{I(2,t)-1}[>\texttt{il}] && \text{(IH)} \\ &\leftrightarrow eret_\sigma^{I(2,t)-1} \wedge /\, jisr_\sigma^{I(2,t)-1} && \text{(definition)} \end{aligned}$$

Note, in the lines above, for any *eret* instruction (i) we clearly have

$$eret_\sigma^i \ \rightarrow \ mca_\sigma^i[10:6] = 0_5.$$

Thus, if the rollback request signal is active

$$rbr_2^t = 1$$

we proceed to show

$$\sigma_1 = 0.$$

From the first part of lemma 75 and definition of the scheduling functions we resp. have

$$ue_3^t \ \rightarrow \ rfull_3^{t+1}$$

and

$$ue_3^t \ \rightarrow \ I(3,t+1) = I(2,t).$$

From the arguments above we conclude

$$rfull_3^{t+1} \wedge (eret_\sigma \wedge /\, jisr_\sigma)^{I(3,t+1)-1}$$

which by definition implies

$$\Sigma(t+1) \neq 3.$$

Using lemma 76 and equation 93 we resp. further conclude

$$\nexists k < 3 : \ rfull_k^{t+1}$$

and

$$\nexists k > 3 : \ rfull_k^{t+1} \wedge (jisr_\sigma \vee eret_\sigma)^{I(k,t+1)-1}$$

which gives the claim.

$$\Sigma(t+1) = 0$$

Otherwise, in case the rollback request signal is inactive

$$rbr_2^t = 0$$

we are to show

$$\sigma_1 = \sigma_0 + 1.$$

Analogous to the previous case we derive

$$rfull_3^{t+1} \wedge jisr_\sigma^{I(3,t+1)-1}$$

which by definition implies

$$\Sigma(t+1) \geq 3.$$

Equation 93 gives

$$\Sigma(t+1) \leq 3$$

and the claim follows.

- $\sigma_0 = 5$. Finally, in this case from lemma 126 we have

  $$rbr_5^t = 1$$

  and we proceed to show

  $$\sigma_1 = 0.$$

  Using lemma 76 we conclude

  $$\nexists k < 6 : \ rfull_k^{t+1}$$

  which gives the claim.

  $$\Sigma(t+1) = 0$$ □

The property of speculation stage Σ below can be easily established following analogous arguments to those in the proof above.

$$\Sigma(t) < R(t) \ \rightarrow \ \Sigma(t+1) = 0 \tag{94}$$

For convenience, we generalize the lemma above as follows.

Lemma 128.

$$\Sigma(t+1) \leq \Sigma(t) + ue_{\Sigma(t)+1}^t$$

Proof of lemma 128. In case

$$\Sigma(t) < R(t)$$

the claim holds trivially by equation 94. Otherwise, for $\Sigma(t) > 0$ the claim follows directly from lemma 127. For $\Sigma(t) = 0$ we proceed to show

$$\Sigma(t+1) \leq ue_1^t.$$

Repeating the arguments presented in the proof of equation 93 we derive

$$\nexists k > 1 : \ rfull_k^{t+1} \wedge (jisr_\sigma \vee eret_\sigma)^{I(k,t+1)-1}$$

which by definition implies

$$\Sigma(t+1) \leq 1.$$

The remaining arguments are analogous to those presented in the proof of lemma 122. □

Properties of Σ_{hard}

Below we show that most of the properties of Σ also hold for hardware speculation stage

$$\Sigma_{hard}(t) = \max\{\Sigma(t), \mu(t)\}.$$

First, using lemma 101 we argue that the rollback request signals are not generated below $\Sigma_{hard}(t)$.

Lemma 129.

$$R(t) \leq \Sigma_{hard}(t)$$

Moreover, from lemmas 102 and 126 we conclude that speculation stage Σ_{hard} stays within interval $[0:5]$.

Lemma 130. *Assume $\Sigma_{hard}(t) = 5$.*

$$ue_6^t \ \rightarrow \ rbr_5^t$$

In the following lemma we derive the value of Σ_{hard} in the next cycle, based on its value and the control signals active in the current cycle.

Lemma 131. *Assume $\Sigma_{hard}(t) = \sigma$ such that $\sigma > 0$.*

$$\overline{ue^t_{\sigma+1}} \rightarrow \Sigma_{hard}(t+1) = \sigma \tag{1}$$

$$ue^t_{\sigma+1} \rightarrow \Sigma_{hard}(t+1) = \begin{cases} 0 & rbr^t_\sigma \\ \sigma+1 & otherwise \end{cases} \tag{2}$$

Proof of lemma 131.1. Given that the stage below σ is not updated in cycle t ($/ue^t_{\sigma+1}$), we split cases on whether $\mu(t)$ is greater, smaller, or equal to $\Sigma(t)$.

- $\Sigma(t) < \mu(t)$. In this case by definition we have

  $$\Sigma_{hard}(t) = \mu(t)$$

  and therefore

  $$\begin{aligned} \Sigma(t) &\leq \Sigma(t)+1 \qquad \text{(lemma 128)} \\ &\leq \mu(t). \end{aligned} \tag{95}$$

  Using the result above we argue as follows.

  $$\begin{aligned} \Sigma_{hard}(t+1) &= \max\{\Sigma(t+1), \mu(t+1)\} \qquad \text{(definition)} \\ &= \max\{\Sigma(t+1), \mu(t)\} \qquad \text{(lemma 103.1)} \\ &= \mu(t) \qquad \text{(equation 95)} \end{aligned}$$

- $\Sigma(t) > \mu(t)$. In this case by definition we have

  $$\Sigma_{hard}(t) = \Sigma(t)$$

  and therefore

  $$\begin{aligned} \mu(t+1) &\leq \mu(t)+1 \qquad \text{(lemma 104)} \\ &\leq \Sigma(t). \end{aligned} \tag{96}$$

  The following arguments are analogous to those presented in the case above.

  $$\begin{aligned} \Sigma_{hard}(t+1) &= \max\{\Sigma(t+1), \mu(t+1)\} \qquad \text{(definition)} \\ &= \max\{\Sigma(t), \mu(t+1)\} \qquad \text{(lemma 127.1)} \\ &= \Sigma(t) \qquad \text{(equation 96)} \end{aligned}$$

- $\Sigma(t) = \mu(t)$. In this case we have

  $$\begin{aligned} \Sigma(t+1) &= \Sigma(t) \qquad \text{(lemmas 127.1, 128)} \\ &= \mu(t) \\ &= \mu(t+1) \qquad \text{(lemmas 103.1, 104)} \end{aligned}$$

  which clearly yields

  $$\Sigma_{hard}(t+1) = \Sigma_{hard}(t).$$ □

Proof of lemma 131.2. Given that the stage below σ is updated in cycle t ($ue^t_{\sigma+1}$), we split cases on whether $\mu(t)$ is greater than $\Sigma(t)$ or not.

- $\Sigma(t) < \mu(t)$. In this case by definition we have

  $$\Sigma_{hard}(t) = \mu(t)$$

  and therefore

  $$\begin{aligned} \Sigma(t+1) &\leq \Sigma(t)+1 \qquad \text{(lemma 128)} \\ &\leq \mu(t). \end{aligned} \tag{97}$$

In case stage σ is rolled-back in cycle t (rbr^t_σ), we clearly have

$$\begin{aligned}\Sigma(t+1) &= 0 \qquad \text{(equation 94)}\\ &= \mu(t+1) \qquad \text{(lemma 103.2)}\end{aligned}$$

which clearly gives

$$\Sigma_{hard}(t+1) = 0.$$

Otherwise, from lemma 102 we conclude

$$\mu(t) < 5$$

and argue as follows.

$$\begin{aligned}\Sigma_{hard}(t+1) &= \max\{\Sigma(t+1), \mu(t+1)\} \qquad \text{(definition)}\\ &= \max\{\Sigma(t+1), \mu(t)+1\} \qquad \text{(lemma 103.2)}\\ &= \mu(t)+1 \qquad \text{(equation 97)}\end{aligned}$$

- $\Sigma(t) \geq \mu(t)$. In this case by definition we have

$$\Sigma_{hard}(t) = \Sigma(t)$$

and therefore

$$\begin{aligned}\mu(t+1) &\leq \mu(t)+1 \qquad \text{(lemma 104)}\\ &\leq \Sigma(t)+1.\end{aligned} \tag{98}$$

From lemma 126 we have

$$\Sigma(t) < 5$$

and the claim follows from the arguments below.

$$\begin{aligned}\Sigma_{hard}(t+1) &= \max\{\Sigma(t+1), \mu(t+1)\} \qquad \text{(definition)}\\ &= \max\{\Sigma(t)+1, \mu(t+1)\} \qquad \text{(lemma 127.2)}\\ &= \Sigma(t)+1 \qquad \text{(equation 98)}\end{aligned}$$

□

Combining lemmas 104 and 128 we easily conclude the following generalization of the result above.

Lemma 132. *Assume* $\Sigma_{hard}(t) = \sigma$.

$$\Sigma_{hard}(t+1) \leq \sigma + ue^t_{\sigma+1}$$

Properties of $\hat{\Sigma}$

In the remainder of this section we show that the corresponding properties (to those of Σ and Σ_{hard}) also hold for

$$\hat{\Sigma}(t) = \max\{\Sigma_{soft}(t), \Sigma_{hard}(t)\}.$$

Clearly, using lemma 129 we argue that the rollback request signals are not generated in stages below $\hat{\Sigma}(t)$.

Lemma 133.

$$R(t) \leq \hat{\Sigma}(t)$$

The value of $\hat{\Sigma}$ in the next cycle is derived in the following lemma.

Lemma 134. *Assume* $\hat{\Sigma}(t) > 0$.

$$\overline{ue^t_{\hat{\Sigma}(t)+1}} \rightarrow \hat{\Sigma}(t+1) = \hat{\Sigma}(t) \tag{1}$$

$$ue^t_{\hat{\Sigma}(t)+1} \rightarrow \hat{\Sigma}(t+1) = \begin{cases} 0 & rbr^t_{\hat{\Sigma}(t)} \wedge \Sigma_{soft}(t) < \Sigma_{hard}(t) \\ \hat{\Sigma}(t)+1 & otherwise \end{cases} \tag{2}$$

Proof of lemma 134.1. Given that the stage below $\hat{\Sigma}(t)$ is not updated in cycle t ($/ue^t_{\hat{\Sigma}(t)+1}$), we split cases on whether $\Sigma_{hard}(t)$ is greater, smaller, or equal to $\Sigma_{soft}(t)$.

- $\Sigma_{soft}(t) < \Sigma_{hard}(t)$. In this case by definition we have

$$\hat{\Sigma}(t) = \Sigma_{hard}(t)$$

and therefore

$$\begin{aligned} \Sigma_{soft}(t+1) &\leq \Sigma_{soft}(t) + 1 \qquad \text{(lemma 122)} \\ &\leq \Sigma_{hard}(t). \end{aligned} \tag{99}$$

Using the result above we proceed to derive the claim as follows.

$$\begin{aligned} \hat{\Sigma}(t+1) &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t+1)\} \qquad \text{(definition)} \\ &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t)\} \qquad \text{(lemma 131.1)} \\ &= \Sigma_{hard}(t) \qquad \text{(equation 99)} \end{aligned}$$

- $\Sigma_{soft}(t) > \Sigma_{hard}(t)$. In this case by definition we have

$$\hat{\Sigma}(t) = \Sigma_{soft}(t)$$

and therefore

$$\begin{aligned} \Sigma_{hard}(t+1) &\leq \Sigma_{hard}(t) + 1 \qquad \text{(lemma 132)} \\ &\leq \Sigma_{soft}(t). \end{aligned} \tag{100}$$

The remaining arguments are analogous to those presented in the case above.

$$\begin{aligned} \hat{\Sigma}(t+1) &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t+1)\} \qquad \text{(definition)} \\ &= \max\{\Sigma_{soft}(t), \Sigma_{hard}(t+1)\} \qquad \text{(lemma 121.1)} \\ &= \Sigma_{soft}(t) \qquad \text{(equation 100)} \end{aligned}$$

- $\Sigma_{soft}(t) = \Sigma_{hard}(t)$. In this case we have

$$\begin{aligned} \Sigma_{soft}(t+1) &= \Sigma_{soft}(t) \qquad \text{(lemmas 121.1, 122)} \\ &= \Sigma_{hard}(t) \\ &= \Sigma_{hard}(t+1) \qquad \text{(lemmas 131.1, 132)} \end{aligned}$$

which clearly gives

$$\hat{\Sigma}(t+1) = \hat{\Sigma}(t). \qquad \square$$

Proof of lemma 134.2. Given that the stage below $\hat{\Sigma}(t)$ is updated in cycle t ($ue^t_{\hat{\Sigma}(t)+1}$), we split cases on whether $\Sigma_{hard}(t)$ is greater than $\Sigma_{soft}(t)$ or not.

- $\Sigma_{soft}(t) < \Sigma_{hard}(t)$. In this case by definition we have

$$\hat{\Sigma}(t) = \Sigma_{hard}(t)$$

and therefore

$$\begin{aligned} \Sigma_{soft}(t+1) &\leq \Sigma_{soft}(t) + 1 \qquad \text{(lemma 122)} \\ &\leq \Sigma_{hard}(t). \end{aligned} \tag{101}$$

In case stage $\hat{\Sigma}(t)$ is rolled-back in cycle t ($rbr^t_{\hat{\Sigma}(t)}$), for stage $\sigma = \Sigma_{soft}(t)$ from the construction of the stall engine we clearly have

$$rbr^t_{\hat{\Sigma}(t)} \rightarrow rbr^t_{\sigma+1}$$

and therefore

$$\begin{aligned}\Sigma_{soft}(t+1) &= 0 \qquad \text{(lemma 121.1)}\\ &= \Sigma_{hard}(t+1) \qquad \text{(lemma 131.2)}\end{aligned}$$

which clearly yields

$$\hat{\Sigma}(t+1) = 0.$$

Otherwise ($/rbr^t_{\hat{\Sigma}(t)}$), from lemma 130 we have

$$\Sigma_{hard}(t) < 5$$

and the claim follows from the arguments below.

$$\begin{aligned}\hat{\Sigma}(t+1) &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t+1)\} \qquad \text{(definition)}\\ &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t)+1\} \qquad \text{(lemma 131.2)}\\ &= \Sigma_{hard}(t)+1 \qquad \text{(equation 101)}\end{aligned}$$

- $\Sigma_{soft}(t) \geq \Sigma_{hard}(t)$. In this case by definition we have

$$\hat{\Sigma}(t) = \Sigma_{soft}(t)$$

and therefore

$$\begin{aligned}\Sigma_{hard}(t+1) &\leq \Sigma_{hard}(t)+1 \qquad \text{(lemma 132)}\\ &\leq \Sigma_{soft}(t)+1. \end{aligned} \tag{102}$$

From equation 88 we know

$$\Sigma_{soft}(t) < 5$$

and the claim follows from the arguments below.

$$\begin{aligned}\hat{\Sigma}(t+1) &= \max\{\Sigma_{soft}(t+1), \Sigma_{hard}(t+1)\} \qquad \text{(definition)}\\ &= \max\{\Sigma_{soft}(t)+1, \Sigma_{hard}(t+1)\} \qquad \text{(lemma 121.2)}\\ &= \Sigma_{soft}(t)+1 \qquad \text{(equation 102)}\end{aligned}$$ □

Finally, combining lemmas 122 and 132 we conclude the following generalization of the result above.

Lemma 135.

$$\hat{\Sigma}(t+1) \leq \hat{\Sigma}(t) + ue^t_{\hat{\Sigma}(t)+1}$$

9.5 Correctness for Memory

All processors that execute memory operations in cycle t we collect into the set

$$PA(t) = \{q \in PS(t) \mid exec(q,t) \wedge mop.5^{q,t}_{\pi}\}.$$

Accordingly, we collect TLBs performing walk extensions in cycle t into the sets

$$TA_Y(t) = \{q \in TS_Y(t) \mid wext(mmu^{q,t}_Y)\}$$

depending on the MMU which is performing the operation. Using the new notation we characterize the stepping numbers y of components q stepped in cycle t.

Lemma 136. *Assume that component q performs a step in cycle t.*

$$s(NS(t)+y).(g,u) = (proc,q) \rightarrow (\ y < sa(t) \leftrightarrow q \in PA(t)\ ) \tag{1}$$

$$s(NS(t)+y).(g,u) = (mmu_Y,q) \rightarrow (\ y < sa(t) \leftrightarrow q \in TA_Y(t)\ ) \tag{2}$$

Proof of lemma 136.1. For processor q accessing memory in cycle t there is an access to the data cache ending in cycle t (lemma 88) and vice versa (lemma 89).

$$q \in PA(t) \;\leftrightarrow\; \exists k : (4q+3,k) \in S_D(t)$$

Since at most one data access ends in the data cache of processor q in cycle t, the claim follows from the definition of the stepping function (p. 229, equation 67). $\square$

Proof of lemma 136.2. For mmu_Y on processor q accessing memory in cycle t there is an access to the corresponding translation cache ending in cycle t (lemma 53). The opposite is true only in case mmu_Y on processor q generates a step in cycle t (lemma 54). As a result, we conclude the following.

$$q \in TA_Y(t) \;\leftrightarrow\; \exists k : (4q+2\mathbb{1}_{\{Y=E\}},k) \in S_Y(t)$$

Analogous to above, since at most one translation access ends in the corresponding translation cache of processor q in cycle t, the claim follows from the definition of the stepping function (p. 229, equation 68). $\square$

Moreover, we argue that the total number of components which are stepped in cycle t while accessing memory equals the total number of stepping accesses.

$$\#PA(t) + \#TA_I(t) + \#TA_E(t) = sa(t).$$

In order to map between the step numbers and numbers of the processor accesses, we introduce the following helper function. For $y < sa(t)$ we define:

$$\begin{aligned} \gamma(0,t) &= \min\{\tilde{y} \mid NE(t) + x(\tilde{y},t) \in seq(S(t))\} \\ \gamma(y,t) &= \min\{\tilde{y} \mid NE(t) + x(\tilde{y},t) \in seq(S(t)) \wedge \tilde{y} > \gamma(y-1,t)\}. \end{aligned}$$

A simple lemma follows from the monotonicity of function x (see p. 198).

Lemma 137. *For $y < sa(t)$ the following holds.*

$$z(y,t) = x(\gamma(y,t),t)$$

In the induction step below (Sect. 9.5.2) we require the following technical lemma.

Lemma 138. *For $k < na(t)$ the following holds.*

$$\forall y < sa(t) : k \neq \gamma(y,t) \;\rightarrow\; xacc'_t[k].(w,cas) = 0_2$$

Proof of lemma 138. First we show the following auxiliary result.

$$n \in seq(E(t) \setminus S_D(t)) \;\rightarrow\; acc'[n].(w,cas) = 0_2 \tag{103}$$

Proof of equation 103. From the assumptions we derive

$$\exists a \in E(t) \setminus S_D(t) \wedge seq(a) = n$$

which allows us to conclude:

$$\begin{aligned} a \in E(t) \setminus S_D(t) &\rightarrow acc(a).(w,cas) = 0_2 \\ &\rightarrow acc'[seq(a)].(w,cas) = 0_2 \\ &\rightarrow acc'[n].(w,cas) = 0_2. \end{aligned}$$ $\square$

Using the result above and monotonicity of function z, we proceed as follows.

$$\begin{aligned} \forall y < sa(t) : k \neq \gamma(y,t) &\rightarrow \forall y < sa(t) : NE(t) + x(k,t) \neq NE(t) + z(y,t) && \text{(lemma 137)} \\ &\rightarrow NE(t) + x(k,t) \in seq(A(t) \setminus S(t)) && \text{(definition)} \\ &\rightarrow acc'[NE(t) + x(k,t)].(w,cas) = 0_2 && \text{(equation 103)} \\ &\rightarrow xacc'_t[k].(w,cas) = 0_2 && \text{(definition)} \end{aligned}$$ $\square$

9.5.1 Matching Processor Accesses with Non-Void Accesses

In this section we identify the accesses registered at various ports (caches) of the memory system in cycle t with the accesses performed by the ISA computation within interval $CS(t)$. Below we begin with data accesses, resp. accesses registered at the data caches.

Data Accesses

First, we argue about the accesses performed by processors on memory operations.

Lemma 139. *Assume that processor q performs a memory access in cycle t.*

$$q \in PA(t) \wedge s(NS(t)+y).(t,u) = (core,q)$$

For access components $Z \in \{a, data, cdata, bw, type\}$ *the following holds.*

$$used(dacc.Z, NS(t)+y) \rightarrow dacc(q, ic(q, NS(t)+y)).Z = zacc'_t[y].Z$$

Proof of lemma 139. Using definition of the stepping function, for some access number k we argue:

$$\begin{aligned} zacc'_t[y].Z &= acc'[NE(t)+z(y,t)].Z \qquad \text{(definition)} \\ &= acc'[seq(4q+3,k)].Z \qquad \text{(lemma 136.1)} \\ &= acc(4q+3,k).Z \qquad \text{(definition)} \\ &= dacc_{\sigma}^{q,i}.Z \qquad \text{(equation 88; IH)} \\ &= dacc(q, ic(q, NS(t)+y)).Z \qquad \text{(lemma 118)} \end{aligned}$$

where by i we denoted the index of instruction executed on processor q in cycle t.

$$\begin{aligned} i &= I(q,6,t) \\ &= I(q,5,t)-1 \qquad \text{(lemma 79)} \end{aligned}$$ □

Moreover, using the software conditions from Sect. 9.2.2 (equation 70) we conclude that the read-only potion of the memory system (ROM) is never written.

Lemma 140. *Assume that processor q performs a memory access in cycle t.*

$$q \in PA(t) \wedge s(NS(t)+y).(t,u) = (core,q)$$

Then the following holds.

$$zacc'_t[y].a < 2^r \rightarrow zacc'_t[y].(w,cas) = 0_2$$

Next, we argue that the processors which do not execute memory operations necessarily perform void accesses.

Lemma 141. *Assume that processor q stepped in cycle t does not access the memory.*

$$q \in PS(t) \setminus PA(t) \wedge s(NS(t)+y).(t,u) = (core,q)$$

Then the corresponding data access is void.

$$void(dacc(q, ic(q, NS(t)+y)))$$

Proof of lemma 141. From the induction hypothesis we derive:

$$\begin{aligned} /mop.5_{\pi}^{q,t} &\rightarrow /mop_{\sigma}^{q,I(q,5,t)-1} \qquad \text{(equation 88; IH)} \\ &\rightarrow /mop_{\sigma}^{q,I(q,6,t)} \qquad \text{(lemma 79)} \\ &\rightarrow void(dacc_{\sigma}^{q,I(q,6,t)}) \qquad \text{(definition)} \\ &\rightarrow void(dacc(q, ic(q, NS(t)+y))) \qquad \text{(lemma 118).} \end{aligned}$$ □

Translation Accesses

First, for the address of the page table entry read by mmu_Y on processor q we derive the following result.

Lemma 142. *Assume that mmu_Y on processor q performs a memory access in cycle t.*

$$q \in TA_Y(t) \wedge s(NS(t)+y).(g,u) = (mmu_Y, q)$$

Then for the page table entry address read by mmu_Y the following holds.

$$ptea(mmu_Y^{q,t}) = ptea(s(NS(t)+y).o)$$

Proof of lemma 142. From the construction we know: mmu_Y^q in cycle t performs extension of the *user walk* iff the nested control automaton of mmu_Y^q resides in state *fetch-pte*$_U$ in cycle t.

$$q \in TA_Y(t) \;\rightarrow\; (\; mmu_Y^{q,t}.\textit{fetch-pte}_U \leftrightarrow wext_U(mmu_Y^{q,t}) \;) \tag{104}$$

Given the equivalence above, we argue simply as follows:

$$\begin{aligned} ptea(mmu_Y^{q,t}) &= \begin{cases} ptea_U(mmu_Y^{q,t}) & wext_U(mmu_Y^{q,t}) \\ ptea_G(mmu_Y^{q,t}) & \text{otherwise} \end{cases} && \text{(definition; equation 104)} \\ &= \begin{cases} ptea(mmu_Y^{q,t}.w_U \circ mmu_Y^{q,t}.w_G) & wext_U(mmu_Y^{q,t}) \\ ptea(mmu_Y^{q,t}.w_G) & \text{otherwise} \end{cases} && \text{(lemma 20)} \\ &= ptea(s(NS(t)+y).o) && \text{(stepping, p. 235).} \end{aligned}$$ □

Using the latter result, we argue about the accesses performed by MMUs on walk extensions.

Lemma 143. *Assume that mmu_Y on processor q performs a memory access in cycle t.*

$$q \in TA_Y(t) \wedge s(NS(t)+y).(g,u) = (mmu_Y, q)$$

For access components $Z \in \{a, bw, type\}$ the following holds.

$$tacc(NS(t)+y).Z = zacc'_t[y].Z$$

Proof of lemma 143. Using definition of the stepping function, for some access number k we argue:

$$\begin{aligned} zacc'_t[y].Z &= acc'[NE(t)+z(y,t)].Z && \text{(definition)} \\ &= acc'[seq(4q+2\mathbb{1}_{\{Y=E\}},k)].Z && \text{(lemma 136.2)} \\ &= acc(4q+2\mathbb{1}_{\{Y=E\}},k).Z && \text{(definition)} \\ &= tacc(s(NS(t)+y)).Z && \text{(lemma 142)} \\ &= tacc(NS(t)+y).Z && \text{(definition, p. 226).} \end{aligned}$$ □

Analogous to lemma 141, we argue that the MMUs which do not perform walk extensions necessarily perform void accesses.

Lemma 144. *Assume that mmu_Y of processor q stepped in cycle t does not access the memory.*

$$q \in TS_Y(t) \setminus TA_Y(t) \wedge s(NS(t)+y).(g,u) = (mmu_Y, q)$$

Then the corresponding translation access is void.

$$void(tacc(NS(t)+y))$$

Proof of lemma 144. Directly from the assumptions we derive:

$$\begin{aligned} /wext(mmu_Y^{q,t}) &\rightarrow winit(mmu_Y^{q,t}) && \text{(assumption)} \\ &\rightarrow s(NS(t)+y).o.t = winit && \text{(stepping)} \\ &\rightarrow void(tacc(NS(t)+y)) && \text{(definition).} \end{aligned}$$ □

9.5.2 Induction Step

For the memory component of ISA we proceed to show the following property.

Lemma 145. *For $y \in [0 : ns(t)]$ we claim*

$$\ell(mc^{NS(t)+y}.m) = \begin{cases} \Delta_M^{\gamma(y,t)}(\ell(mc^{NS(t)}.m), xacc'_t) & y < sa(t) \\ \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t) & \textit{otherwise} \end{cases}$$

Proof of lemma 145. By induction on y. For convenience we introduce the following shorthand.

$$acc_\sigma = \begin{cases} dacc(q, ic(q, NS(t)+y)) & s(NS(t)+y).(g,u) = (proc, q) \\ tacc(NS(t)+y) & s(NS(t)+y).(g,u) = (mmu_Y, q) \end{cases}$$

For the base case, if stepping accesses are performed in cycle t ($sa(t) > 0$), we have

$$\begin{aligned} \Delta_M^{\gamma(0,t)}(\ell(mc^{NS(t)}.m), xacc'_t) &= \Delta_M^{\gamma(0,t)}(\ell(mc^{NS(t)}.m), xacc'_t[0 : \gamma(0,t)-1]) \\ &= \ell(mc^{NS(t)+0}.m) \qquad \text{(lemma 138).} \end{aligned}$$

Otherwise, if no stepping accesses are performed in cycle t ($sa(t) = 0$), we have

$$\begin{aligned} \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t) &= \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t[0 : na(t)-1]) \\ &= \ell(mc^{NS(t)+0}.m) \qquad \text{(lemma 138).} \end{aligned}$$

For the induction step from y to $y+1$ we split cases on the value of y.

- $y < sa(t)$. From lemmas 139 and 143 we have

$$acc_\sigma = zacc'_t[y] \tag{105}$$

  and argue using the induction hypothesis for $y < sa(t)$:

$$\begin{aligned} \ell(mc^{NS(t)+y+1}.m) &= \Delta_M(\ell(mc^{NS(t)+y}.m), acc_\sigma) \qquad \text{(definition)} \\ &= \Delta_M(\Delta_M^{\gamma(y,t)}(\ell(mc^{NS(t)}.m), xacc'_t), zacc'_t[y]) \qquad \text{(ind. hypothesis; equation 105)} \\ &= \Delta_M(\Delta_M^{\gamma(y,t)}(\ell(mc^{NS(t)}.m), xacc'_t[0 : \gamma(y,t)-1]), xacc'_t[\gamma(y,t)]) \qquad \text{(lemma 137)} \\ &= \Delta_M^{\gamma(y,t)+1}(\ell(mc^{NS(t)}.m), xacc'_t) \\ &= \begin{cases} \Delta_M^{\gamma(y+1,t)}(\ell(mc^{NS(t)}.m), xacc'_t) & y+1 < sa(t) \\ \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t) & \text{otherwise} \end{cases} \qquad \text{(lemma 138).} \end{aligned}$$

- $y \geq sa(t)$. From lemmas 141 and 144 we have

$$void(acc_\sigma) \tag{106}$$

  and argue using the induction hypothesis for $y \geq sa(t)$:

$$\begin{aligned} \ell(mc^{NS(t)+y+1}.m) &= \Delta_M(\ell(mc^{NS(t)+y}.m), acc_\sigma) \qquad \text{(definition)} \\ &= \ell(mc^{NS(t)+y}.m) \qquad \text{(equation 106)} \\ &= \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t) \qquad \text{(induction hypothesis).} \end{aligned}$$ □

Using the results of this section we complete the induction step for the memory part with almost no effort. We argue as follows:

$$\begin{aligned} m(h_\pi^{t+1}) &= \Delta_M^{na(t)}(m(h_\pi^t), xacc'_t) \qquad \text{(lemma 91)} \\ &= \Delta_M^{na(t)}(\ell(mc^{NS(t)}.m), xacc'_t) \qquad \text{(IH)} \\ &= \ell(mc^{NS(t)+ns(t)}.m) \qquad \text{(lemma 145)} \\ &= \ell(mc^{NS(t+1)}.m) \qquad \text{(definition).} \end{aligned}$$

9.5.3 Outputs to Accesses

Next we show that outputs of the cache memory system to accesses performed by the processor cores and MMUs match the outputs observed in the ISA. The processor cores access the memory system to fetch instructions and to execute load or CAS operations, whereas the MMUs — to perform walk extensions.

Data Access

First, we argue about the outputs to the accesses performed by processors executing load or CAS operations.

Lemma 146. *Assume that processor q stepped in cycle t executes a load or a CAS operation.*

$$(l.5_{\pi}^{q,t} \vee cas.5_{\pi}^{q,t}) \wedge s(NS(t)+y).(t,u) = (core,q)$$

Then the following holds.

$$pdout(4q+3)^{t} = dmout_{\sigma}^{q,I(q,6,t)}$$

Proof of lemma 146. For instruction executed on processor q in cycle t by lemma 118 we have

$$i = I(q,6,t) = ic(q,NS(t)+y)$$

while for the line address of the corresponding data access, for some access number k we derive:

$$\begin{aligned} a = dacc(q,i).a &= zacc'_{t}[y].a \qquad \text{(lemma 139)} \\ &= acc'[NE(t)+z(y,t)].a \qquad \text{(definition)} \\ &= acc'[seq(4q+3,k)].a \qquad \text{(lemma 136.1; equation 67).} \end{aligned}$$

Then the output to the data access of the specification would be:

$$\begin{aligned} dmout_{\sigma}^{q,i} &= dataout(\ell(mc^{pseq(q,i)}.m),dacc(q,i)) \qquad \text{(definition)} \\ &= \ell(mc^{pseq(q,i)}.m)(a) \qquad \text{(definition)} \\ &= \ell(mc^{NS(t)+y}.m)(a) \qquad \text{(lemma 110)} \\ &= \Delta_{M}^{\gamma(y,t)}(\ell(mc^{NS(t)}.m),xacc'_{t})(a) \qquad \text{(lemma 145)} \\ &= \Delta_{M}^{\gamma(y,t)}(m(h_{\pi}^{t}),xacc'_{t})(a) \qquad \text{(IH)} \\ &= M_{\gamma(y,t)}(a) \qquad \text{(definition, p. 198).} \end{aligned}$$

Using definition of the stepping function, for some access number k we proceed to show for the output of the data cache:

$$\begin{aligned} M_{\gamma(y,t)}(a) &= \Delta_{M}^{x(\gamma(y,t),t)}(m(h_{\pi}^{t}),acc'_{t})(a) \qquad \text{(lemma 90)} \\ &= \Delta_{M}^{x(\gamma(y,t),t)}(\Delta_{M}^{NE(t)}(m(h_{\pi}^{0}),acc'),acc'_{t})(a) \qquad \text{(lemma 51)} \\ &= \Delta_{M}^{NE(t)+z(y,t)}(m(h_{\pi}^{0}),acc')(a) \qquad \text{(lemma 137)} \\ &= \Delta_{M}^{seq(4q+3,k)}(m(h_{\pi}^{0}),acc')(a) \qquad \text{(lemma 136.1; equation 67)} \\ &= pdout(4q+3)^{t} \qquad \text{(lemma 52).} \end{aligned}$$

□

Instruction Fetch

Next, we argue about the outputs to the accesses performed by processors fetching instructions.

Lemma 147. *Assume that stage* 2 *on processor q is updated "below"* $\vec{\Sigma}$ *in cycle t.*

$$\vec{\Sigma}(q,t) < (1,2) \wedge ue_{2}^{q,t}$$

Then the following holds.

$$pdout(4q+1)^{t} = imout_{\sigma}^{q,I(q,2,t)}$$

Proof of lemma 147. The instruction fetched on processor q in cycle t is abbreviated as

$$i = I(q,2,t)$$

whereas for the line address of the corresponding instruction fetch access we derive:

$$\begin{aligned} a = iacc(q,i).a &= pma_{I\ \sigma}^{q,i}.l && \text{(definition)} \\ &= pmaI.1_{\pi}^{q,t}.l && \text{(IH).} \end{aligned}$$

Note, by definition of $\vec{\Sigma}$ we know that software conditions for instruction fetch are met for instruction i on processor q.

$$SC_{code}(q,i)$$

We proceed to show that the memory line at address a is the same in configurations $NS(t)$ and $pseq(q,i)$, in which instruction i is executed.

$$\ell(mc^{NS(t)}.m)(a) = \ell(mc^{pseq(q,i)}.m)(a) \tag{107}$$

Proof of equation 107. From the definition (equation 69) we conclude that instructions in steps

$$n \in [pseq(q,i-5)+1 : pseq(q,i+1)-1]$$

do not modify the ISA memory at line address a.

$$\ell(mc^{n}.m)(a) = \ell(mc^{pseq(q,i+1)}.m)(a)$$

Therefore, it suffices to show that

$$NS(t), pseq(q,i) \in [pseq(q,i-5)+1 : pseq(q,i+1)]$$

which follows from lemma 119 and monotonicity of function *pseq* resp. □

Then the output to the instruction fetch access of the specification would be:

$$\begin{aligned} imout_{\sigma}^{q,i} &= dataout(\ell(mc^{pseq(q,i)}.m), iacc(q,i)) && \text{(definition)} \\ &= \ell(mc^{pseq(q,i)}.m)(a) && \text{(definition)} \\ &= \ell(mc^{NS(t)}.m)(a) && \text{(equation 107).} \end{aligned}$$

In order to complete the proof we require one more auxiliary result.

$$n \in seq(E(t)) \wedge acc'[n].a = a \ \rightarrow \ acc'[n].(w, cas) = 0_2 \tag{108}$$

Proof of equation 108. By contradiction. From the assumptions and equation 103 we derive:

$$\exists d \in seq(S_D(t)) \wedge seq(d) = n.$$

From the definition of S for some processor q', access k, and step number y we have:

$$d = (4q'+3, k) \wedge seq(d) = NE(t) + z(y,t).$$

From the definition of the stepping function we conclude

$$y < sa(t) \wedge s(NS(t)+y).(t,u) = (core, q')$$

which by part (1) of lemma 136 implies

$$q' \in PA(t).$$

Using the arguments above we can apply lemma 139 and first obtain

$$\begin{aligned}dacc(q',ic(q',NS(t)+y)).(w,cas) &= zacc'_t[y].(w,cas)\\ &= acc'[NE(t)+z(y,t)].(w,cas)\\ &= acc'[seq(d)].(w,cas)\\ &= acc'[n].(w,cas) \neq 0_2\end{aligned}$$

and then, analogously obtain

$$dacc(q',ic(q',NS(t)+y)).a = acc'[n].a = a.$$

Therefore, for ISA step

$$m = NS(t)+y \in [NS(t) : NS(t+1)-1]$$

we conclude

$$write(mc^m, s(m)) \wedge pma_E(mc^m, s(m)).l = a$$

which according to lemma 119 violates the software conditions (equation 69). □

In turn, for the output of the instruction cache and some access number k we show:

$$\begin{aligned}pdout(4q+1)^t &= \Delta_M^{seq(4q+1,k)}(m(h_\pi^0), acc'_t)(a) \qquad \text{(lemma 52)}\\ &= \Delta_M^{NE(t)+n_I}(m(h_\pi^0), acc')(a) \qquad (\text{where } n_I \le ne(t))\\ &= \Delta_M^{NE(t)}(m(h_\pi^0), acc')(a) \qquad \text{(equation 108)}\\ &= m(h_\pi^t)(a) \qquad \text{(lemma 51).}\end{aligned}$$

Thus, the result follows directly from the induction hypothesis (IH). □

Translation Access

Finally, we argue about the outputs to the accesses performed by MMUs extending walks.

Lemma 148. *Assume that mmu_Y on processor q performs a walk extension in cycle t.*

$$q \in TA_Y(t) \wedge s(NS(t)+y).(g,u) = (mmu_Y, q).$$

Then the following holds.

$$pdout(4q+2\mathbb{1}_{\{Y=E\}})^t = tmout(NS(t)+y)$$

Proof of lemma 148. Given that for the line address of the translation access, for some access number k we have

$$\begin{aligned}a = tacc(NS(t)+y).a &= zacc'_t[y].a \qquad \text{(lemma 143)}\\ &= acc'[NE(t)+z(y,t)].a \qquad \text{(definition)}\\ &= acc'[seq(4k+2\mathbb{1}_{\{Y=E\}},k)].a \qquad \text{(lemma 136.2; equation 68)}\end{aligned}$$

the output to the translation access of the specification would be:

$$\begin{aligned}tmout(NS(t)+y) &= dataout(\ell(mc^{NS(t)+y}.m), tacc(NS(t)+y)) \qquad \text{(definition)}\\ &= \ell(mc^{NS(t)+y}.m)(a) \qquad \text{(definition)}\\ &= \Delta_M^{\gamma(y,t)}(\ell(mc^{NS(t)}.m), xacc'_t)(a) \qquad \text{(lemma 145)}\\ &= \Delta_M^{\gamma(y,t)}(m(h_\pi^t), xacc'_t)(a) \qquad \text{(IH)}\\ &= M_{\gamma(y,t)}(a) \qquad \text{(definition).}\end{aligned}$$

For the output of the corresponding cache and some access number k we show:

$$\begin{aligned}M_{\gamma(y,t)}(a) &= \Delta_M^{x(\gamma(y,t),t)}(m(h_\pi^t), acc'_t)(a) \qquad \text{(lemma 90)}\\ &= \Delta_M^{x(\gamma(y,t),t)}(\Delta_M^{NE(t)}(m(h_\pi^0), acc'), acc'_t)(a) \qquad \text{(lemma 51)}\\ &= \Delta_M^{NE(t)+z(y,t)}(m(h_\pi^0), acc')(a) \qquad \text{(lemma 137)}\\ &= \Delta_M^{seq(4q+2\mathbb{1}_{\{Y=E\}},k)}(m(h_\pi^0), acc')(a) \qquad \text{(lemma 136.2; equation 68)}\\ &= pdout(4q+2\mathbb{1}_{\{Y=E\}})^t \qquad \text{(lemma 52).}\end{aligned}$$

□

9.6 Correctness for Pipeline Registers

First in this section we present the arguments necessary to perform the induction step for the pipeline registers (Sects. 9.6.1–9.6.2). To show that the registers are simulated correctly, we split cases on the values of speculation stage $\hat{\Sigma}$ in cycles t and $t+1$. The latter case split is as follows:

- $\hat{\Sigma}(t+1) = \hat{\Sigma}(t) = 0$; value of $\hat{\Sigma}$ remains zero (Sect. 9.6.3),
- $\hat{\Sigma}(t+1) = \hat{\Sigma}(t)+1$; value of $\hat{\Sigma}$ increases ($ue_{\hat{\Sigma}(t)+1}$, Sect. 9.6.4), and
- $\hat{\Sigma}(t+1) = \hat{\Sigma}(t) \neq 0$; (non-zero) value of $\hat{\Sigma}$ remains unchanged ($/ue_{\hat{\Sigma}(t)+1}$),
- $\hat{\Sigma}(t+1) = 0 \wedge \hat{\Sigma}(t) = 2$; value of $\hat{\Sigma}$ resets to zero on *pcres* (ue_3, see Sect. 9.6.5),
- $\hat{\Sigma}(t+1) = 0 \wedge \hat{\Sigma}(t) = 5$; value of $\hat{\Sigma}$ resets to zero on *jisr* (ue_6, Sect. 9.6.6).

Note, in the last two cases according to lemma 134 we have $rbr_{\hat{\Sigma}(t)}$ and

$$\Sigma_{soft}(t) < \Sigma_{hard}(t).$$

9.6.1 Speculation on SPR Content

Recall from Chap. 8 that the machine mode is i) (speculatively) used in the upper pipeline stages (Sect. 8.1.5) and ii) written on interrupts and returns from exceptions in the memory stage (Sect. 8.2.1). In this section we justify our speculative usage of the machine mode. In the following lemma we assume the absence of interrupts in the ISA computation as well as the absence of a legal *eret* instruction in the pipeline.

Lemma 149. *For pipelines such that*

$$\hat{\Sigma}(t) = 0 \quad \textit{and} \quad \forall j \in [3:5] : /(rfull_j^t \wedge iret_j^t)$$

we claim the following holds for stages $k \leq 6$:

$$mode(t) = mode_\sigma^{I(k,t)} \tag{1}$$

$$asid(t) = asid_\sigma^{I(k,t)} \tag{2}$$

Proof of lemma 149.1. From the definition of Σ we know

$$\forall j \in [1:5] : /(rfull_j^t \wedge jisr_\sigma^{I(j,t)-1}) \tag{109}$$

and

$$\forall j \in [1:2] : /(rfull_j^t \wedge eret_\sigma^{I(j,t)-1}).$$

From the assumptions for stages $j \in [3:5]$ we derive

$$\begin{aligned} /(rfull_j^t \wedge iret_j^t) &\rightarrow /(rfull_j^t \wedge eret.j_\pi^t \wedge ca.j_\pi^t[\texttt{>il}]) \qquad \text{(definition)} \\ &\rightarrow (rfull_j^t \wedge eret_\sigma^{I(j,t)-1} \rightarrow jisr_\sigma^{I(j,t)-1}) \qquad \text{(IH)} \\ &\rightarrow /(rfull_j^t \wedge eret_\sigma^{I(j,t)-1}) \qquad \text{(equation 109).} \end{aligned}$$

Altogether we clearly have

$$\nexists j \in [1:5] : rfull_j^t \wedge (jisr_\sigma^{I(j,t)-1} \vee eret_\sigma^{I(j,t)-1})$$

or using lemma 79:

$$\nexists j \in [2:6] : rfull_{j-1}^t \wedge (jisr_\sigma^{I(j,t)} \vee eret_\sigma^{I(j,t)}). \tag{110}$$

From the induction hypothesis we also know

$$mode(t) = mode_\sigma^{I(6,t)}$$

therefore it suffices to prove

$$mode_\sigma^{I(6,t)} = mode_\sigma^{I(6-\ell,t)}$$

by induction on $\ell \le 5$. For the base case ($\ell = 0$) there is nothing to show. For the induction step from ℓ to $\ell+1 \le 5$ we argue as follows.

$$\begin{aligned}
mode_\sigma^{I(6-\ell-1,t)} &= \begin{cases} mode_\sigma^{I(6-\ell,t)+1} & rfull_{6-\ell-1}^t \\ mode_\sigma^{I(6-\ell,t)} & \text{otherwise} \end{cases} && \text{(lemma 79)} \\
&= \begin{cases} mode_\sigma^{I(6-\ell,t)+1} & /jisr_\sigma^{I(6-\ell,t)} \wedge /eret_\sigma^{I(6-\ell,t)} \\ mode_\sigma^{I(6-\ell,t)} & \text{otherwise} \end{cases} && \text{(equation 110)} \\
&= mode_\sigma^{I(6-\ell,t)} && \text{(specification)} \\
&= mode_\sigma^{I(6,t)} && \text{(induction hypothesis)}
\end{aligned}$$

□

Proof of lemma 149.2. Recall, in Sect. 3.2 we defined the ASID in configuration c as

$$asid(c) = \begin{cases} vmid(c) \circ prid(c) & user(c) \\ vmid(c) \circ 0_8 & guest(c) \\ 0_{12} & host(c). \end{cases}$$

Having lemma 149.1, on the level of host there is clearly nothing to show. For the remaining levels of execution we need to show the following.

$$\begin{aligned}
mode(t) \ne \texttt{host} &\to & vmid(t) &= vmid_\sigma^{I(k,t)} \\
mode(t) = \texttt{user} &\to & prid(t) &= prid_\sigma^{I(k,t)}
\end{aligned}$$

Using the induction hypothesis we rewrite the latter as follows.

$$\begin{aligned}
mode_\sigma^{I(6,t)} \ne \texttt{host} &\to & vmid_\sigma^{I(6,t)} &= vmid_\sigma^{I(k,t)} \\
mode_\sigma^{I(6,t)} = \texttt{user} &\to & prid_\sigma^{I(6,t)} &= prid_\sigma^{I(k,t)}
\end{aligned}$$

Next we argue along the proof lines of lemma 149.1 and first obtain that there are no illegal instructions in truly full stages $j \in [2:6]$.

$$\nexists j \in [2:6] : rfull_{j-1}^t \wedge ill_\sigma^{I(j,t)}$$

Using lemma 149.1 for the execution mode, for the virtual machine ID we derive

$$mode_\sigma^{I(6,t)} \ne \texttt{host} \to \nexists j \in [2:6] : rfull_{j-1}^t \wedge movg2s_\sigma^{I(j,t)} \wedge (xad_\sigma^{I(j,t)} = \texttt{mode})$$

and for the process ID we derive

$$mode_\sigma^{I(6,t)} = \texttt{user} \to \nexists j \in [2:6] : rfull_{j-1}^t \wedge movg2s_\sigma^{I(j,t)} \wedge (xad_\sigma^{I(j,t)} = \texttt{nmode}).$$

For both IDs one can easily complete the proof by induction on $\ell \le 5$ by showing

$$\begin{aligned}
mode_\sigma^{I(6,t)} \ne \texttt{host} &\to & vmid_\sigma^{I(6,t)} &= vmid_\sigma^{I(6-\ell,t)} \\
mode_\sigma^{I(6,t)} = \texttt{user} &\to & prid_\sigma^{I(6,t)} &= prid_\sigma^{I(6-\ell,t)}.
\end{aligned}$$

□

In the next sections we not always can apply the result above, at least not for all stages. For that reason we proceed to limit the scope of lemma 149, which allows us to weaken its assumptions.

Lemma 150. *For stages $k \in [3:6]$ such that*

$$\vec{\Sigma}(t) < (k-1,k) \wedge rfull_{k-1}^t$$

we claim the following holds:

$$mode(t) = mode_\sigma^{I(k,t)} \tag{1}$$

$$asid(t) = asid_\sigma^{I(k,t)} \tag{2}$$

Proof of lemma 150.1. From the definition of Σ we know

$$\forall j \in [k:5] : /(rfull_j^t \wedge jisr_\sigma^{I(j,t)-1}) \tag{111}$$

and from the assumptions for stages $j \in [k:5]$ we derive

$$\begin{aligned} rfull_{k-1}^t &\rightarrow /(rfull_j^t \wedge iret_j^t) \qquad \text{(lemma 77.1)} \\ &\rightarrow /(rfull_j^t \wedge eret.j_\pi^t \wedge ca.j_\pi^t[>\texttt{il}]) \qquad \text{(definition)} \\ &\rightarrow (rfull_j^t \wedge eret_\sigma^{I(j,t)-1} \rightarrow jisr_\sigma^{I(j,t)-1}) \qquad \text{(IH)} \\ &\rightarrow /(rfull_j^t \wedge eret_\sigma^{I(j,t)-1}) \qquad \text{(equation 111).} \end{aligned}$$

Altogether we clearly have

$$\nexists j \in [k:5] : rfull_j^t \wedge (jisr_\sigma^{I(j,t)-1} \vee eret_\sigma^{I(j,t)-1})$$

or using lemma 79:

$$\nexists j \in [k+1:6] : rfull_{j-1}^t \wedge (jisr_\sigma^{I(j,t)} \vee eret_\sigma^{I(j,t)}).$$

From the induction hypothesis we also know

$$mode(t) = mode_\sigma^{I(6,t)}$$

therefore it suffices to prove

$$mode_\sigma^{I(6,t)} = mode_\sigma^{I(6-\ell,t)}$$

by induction on $\ell \leq 6-k$. In order to complete the proof one can literally follow the corresponding lines in the proof of lemma 149.1. □

The proof of lemma 150.2 is similar to the proof of lemma 149.2, and therefore is omitted.

9.6.2 Matching MMU Outputs

Below we argue that in the absence of interrupts both in the hardware and in the ISA computation, the walks output by the two MMUs provide matching translations for the corresponding translation requests.

Lemma 151. *Assume that the address translation is enabled ($/host(t)$).*

$$\hat{\Sigma}(t) = 0 \wedge mca(1)^t[>\texttt{mf}] \wedge ue_1^t \;\rightarrow\; match(trq_{I\;\sigma}^{I(1,t)}, mmu_I^t.wout) \tag{1}$$

$$mop.4_\pi^t \wedge \vec{\Sigma}(t) < (4,5) \wedge mca(5)^t[>\texttt{mm}] \wedge ue_5^t \;\rightarrow\; match(trq_{E\;\sigma}^{I(5,t)}, mmu_E^t.wout) \tag{2}$$

From the specification of MMUs we know that the walks provided to the processor, and thus passed to the specification machine, are either faulty or complete.

$$f(mmu_Y^t.wout) \vee mmu_Y^t.wout.\ell[0]$$

Proof of lemma 151.1. For the walk output of mmu_I we clearly have

$$\begin{aligned} mmu_I^t.wout.upa &= mmu_I^t.upa \qquad \text{(specification)} \\ &= asid(t) \circ ia_\pi^t.pa \qquad \text{(interconnect)} \\ &= asid_\sigma^{I(1,t)} \circ ia_\pi^t.pa \qquad \text{(lemma 149.2)} \\ &= asid_\sigma^{I(1,t)} \circ ia_\sigma^{I(1,t)}.pa \qquad \text{(induction hypothesis).} \end{aligned}$$

Note, in the proof lines above we can apply lemma 149 since, again, here we consider only those cycles t in which stage $k=1$ is updated. From this we immediately derive the absence of a legal *eret* instruction anywhere in the pipeline.

$$ue_1^t \;\rightarrow\; /haz_1^t \;\rightarrow\; /drain(t)$$ □

Proof of lemma 151.2. For the walk output of mmu_E we analogously derive

$$\begin{aligned} mmu_E^t.wout.upa &= mmu_E^t.upa && \text{(specification)} \\ &= asid(t) \circ ea.4_\pi^t.pa && \text{(interconnect)} \\ &= asid_\sigma^{I(5,t)} \circ ea.4_\pi^t.pa && \text{(lemma 150.2)} \\ &= asid_\sigma^{I(5,t)} \circ ea_\sigma^{I(5,t)}.pa && \text{(induction hypothesis).} \end{aligned}$$

Again, since we consider only those cycle t in which stage $k = 5$ is updated, we immediately obtain that stage 4 has a real full bit in cycle t ($rfull_4^t$). This justifies application of lemma 150 above. □

9.6.3 Regular Execution

In this section we present the correctness arguments covering execution of most of the instructions given that

$$\hat{\Sigma}(t+1) = \hat{\Sigma}(t) = 0.$$

First we are to show the following:

- for the instruction address
$$ia_\pi^t = ia_\sigma^{I(1,t)}$$
- for the physical memory addresses
$$\begin{aligned} pma_{I\ \pi}^t &= pma_{I\ \sigma}^{I(1,t)} \\ pma_{E\ \pi}^t &= pma_{I\ \sigma}^{I(5,t)} \end{aligned}$$
- for the instruction fetched
$$I(h_\pi^t) = I_\sigma^{I(2,t)}$$
- for the output of the memory access
$$mout(h_\pi^t) = mout_\sigma^{I(6,t)}$$

Instruction address

We split cases on the number n of real full stages above the ID stage:

$$n(t) = \sum_{1 \le k < 3} rfull_k^t.$$

- $n(t) = 0$.
$$ia_\pi^t = ddpc_\pi^t = ddpc_\sigma^{I(3,t)} = ddpc_\sigma^{I(1,t)} = ia_\sigma^{I(1,t)}$$
- $n(t) = 1 \wedge full_1^t \wedge /rbp_1^t$.
$$ia_\pi^t = dpc_\pi^t = dpc_\sigma^{I(3,t)} = dpc_\sigma^{I(2,t)} = ddpc_\sigma^{I(2,t)+1} = ddpc_\sigma^{I(1,t)} = ia_\sigma^{I(1,t)}$$
- $n(t) = 1 \wedge full_2^t \wedge /rbp_2^t$.
$$ia_\pi^t = dpc_\pi^t = dpc_\sigma^{I(3,t)} = ddpc_\sigma^{I(3,t)+1} = ddpc_\sigma^{I(2,t)} = ddpc_\sigma^{I(1,t)} = ia_\sigma^{I(1,t)}$$
- $n(t) = 2$.
$$ia_\pi^t = pc_\pi^t = pc_\sigma^{I(3,t)} = dpc_\sigma^{I(3,t)+1} = dpc_\sigma^{I(2,t)} = ddpc_\sigma^{I(2,t)+1} = ddpc_\sigma^{I(1,t)} = ia_\sigma^{I(1,t)}$$

Physical Memory Addresses

In case the address translation is used, we show

$$\begin{aligned} pma_{I\ \pi}^{t} &= mmu_I^t.wout.ba \circ ia_\pi^t.po && \text{(interconnect)} \\ &= mmu_I^t.wout.ba \circ ia_\sigma^{I(1,t)}.po && \text{(induction hypothesis)} \\ &= tma(asid_\sigma^{I(1,t)} \circ ia_\sigma^{I(1,t)}, mmu_I^t.wout) && \text{(lemma 151.1)} \\ &= tma(asid_\sigma^{I(1,t)} \circ ia_\sigma^{I(1,t)}, w_{I\ \sigma}^{I(1,t)}) && \text{(equation 73)} \\ &= pma_{I\ \sigma}^{I(1,t)} && \text{(definition)} \end{aligned}$$

and

$$\begin{aligned} pma_{E\ \pi}^{t} &= mmu_E^t.wout.ba \circ ea.4_\pi^t.po && \text{(interconnect)} \\ &= mmu_E^t.wout.ba \circ ea_\sigma^{I(1,t)}.po && \text{(induction hypothesis)} \\ &= tma(asid_\sigma^{I(5,t)} \circ ea_\sigma^{I(5,t)}, mmu_E^t.wout) && \text{(lemma 151.2)} \\ &= tma(asid_\sigma^{I(5,t)} \circ ea_\sigma^{I(5,t)}, w_{E\ \sigma}^{I(5,t)}) && \text{(equation 74)} \\ &= pma_{E\ \sigma}^{I(5,t)} && \text{(definition).} \end{aligned}$$

In case the address translation is not used, we have

$$pma_{I\ \pi}^{t} = ia_\pi^t = ia_\sigma^{I(1,t)} = pma_{I\ \sigma}^{I(1,t)}$$

and

$$pma_{E\ \pi}^{t} = ea.4_\pi^t = ea_\sigma^{I(5,t)} = pma_{E\ \sigma}^{I(5,t)}.$$

For the destination registers (in case they are updated) we now easily derive:

$$pmaI.1_\pi^{t+1} = pma_{I\ \pi}^{t} = pma_{I\ \sigma}^{I(1,t)} = pma_{I\ \sigma}^{I(1,t+1)-1}$$

and

$$pmaE.5_\pi^{t+1} = pma_{E\ \pi}^{t} = pma_{E\ \sigma}^{I(5,t)} = pma_{E\ \sigma}^{I(5,t+1)-1}.$$

Instruction Fetched

For the fetched instruction we show:

$$\begin{aligned} I(h_\pi^t) &= \begin{cases} pdout(4q+1)_H^t & pmaI.1_\pi^t[2] \\ pdout(4q+1)_L^t & \text{otherwise} \end{cases} && \text{(interconnect)} \\ &= \begin{cases} imout_{\sigma\ H}^{q,I(q,2,t)} & pma_{I\ \sigma}^{I(q,2,t)}[2] \\ imout_{\sigma\ L}^{q,I(q,2,t)} & \text{otherwise} \end{cases} && \text{(lemma 147)} \\ &= I_\sigma^{q,I(q,2,t)} && \text{(lemma 7).} \end{aligned}$$

And for the instruction register (in case it is updated) we now easily derive:

$$I_\pi^{t+1} = I(h_\pi^t) = I_\sigma^{q,I(q,2,t)} = I_\sigma^{q,I(q,2,t+1)-1}.$$

Output of Memory Access

For the output of the memory access we show:

$$\begin{aligned} mout(h_\pi^t) &= pdout(4q+3)^t && \text{(interconnect)} \\ &= dmout_\sigma^{q,I(q,6,t)} && \text{(lemma 146)} \\ &= mout_\sigma^{q,I(q,6,t)} && \text{(definition).} \end{aligned}$$

And for the pipeline register (in case it is updated) we now easily derive:

$$mout.6_\pi^{t+1} = mout(h_\pi^t) = mout_\sigma^{q,I(q,6,t)} = mout_\sigma^{q,I(q,6,t+1)-1}.$$

Forwarding for A, B, and S

Forwarding for registers A and B stays literally the same as in [KMP14], except that now it has to be performed over more pipeline stages. For register S we argue as follows. For address

$$rs = rs^t_\pi = rs_\sigma^{I(3,t)}$$

and special purpose register

$$sprx^t_\pi = spr^t_\pi(rs) = spr_\sigma^{I(6,t)}(rs) = sprx_\sigma^{I(6,t)}$$

we derive:

$$\begin{aligned}
S(h^t_\pi) &= \begin{cases} C.4^t_\pi.in & tops[3]^t \\ C.k^t_\pi & tops[k]^t \wedge k > 3 \\ sprx^t_\pi & \text{otherwise} \end{cases} && \text{(interconnect)} \\
&= \begin{cases} C_\sigma^{I(k,t)} & \min\{j \mid movg2s_\sigma^{I(j,t)} \wedge (xad_\sigma^{I(j,t)} = rs)\} = k \in [4:6] \\ sprx_\sigma^{I(6,t)} & \text{otherwise} \end{cases} && \text{(IH)} \\
&= S_\sigma^{I(3,t)} \quad \text{(specification).}
\end{aligned}$$

Therefore, for the S register we easily obtain:

$$S^{t+1}_\pi = S(h^t_\pi) = S_\sigma^{I(3,t)} = S_\sigma^{I(3,t+1)-1}.$$

Typical Pipeline Registers

Obviously, we consider two cases. If register R in stage k is updated in cycle t (ue^t_k), in the spirit of [KMP14] we first have to argue that all signals *used* for computation of the register's input are correct; then, correctness of the register update follows, basically, from the register semantics. For better presentation, let us consider the update of the program counters, the visible registers in stage 3. For the dpc and $ddpc$ resp. the arguments are trivial:

$$dpc^{t+1}_\pi = pc^t_\pi = pc_\sigma^{I(3,t)} = dpc_\sigma^{I(3,t)+1} = dpc_\sigma^{I(3,t+1)}$$

and

$$ddpc^{t+1}_\pi = dpc^t_\pi = dpc_\sigma^{I(3,t)} = ddpc_\sigma^{I(3,t)+1} = ddpc_\sigma^{I(3,t+1)}.$$

In turn, correctness of the pc update hinges on correct computation of the *nextpc*:

$$pc^{t+1}_\pi = nextpc^t_\pi \overset{!}{=} nextpc_\sigma^{I(3,t)} = pc_\sigma^{I(3,t)+1} = pc_\sigma^{I(3,t+1)}.$$

In order to establish correctness of the *nextpc* computation

$$\begin{aligned}
nextpc^t_\pi &= \begin{cases} btarget^t_\pi & jbtaken^t_\pi \\ pc^t_\pi +_{32} 4_{32} & \text{otherwise} \end{cases} && \text{(interconnect)} \\
&\overset{!}{=} \begin{cases} btarget_\sigma^{I(3,t)} & jbtaken_\sigma^{I(3,t)} \\ pc_\sigma^{I(3,t)} +_{32} 4_{32} & \text{otherwise} \end{cases} && \text{(IH)} \\
&= nextpc_\sigma^{I(3,t)} \quad \text{(definition)}
\end{aligned}$$

we show correctness for signals *jbtaken*, which is always used, and *btarget*, which is used only if a jump or branch instruction is executed.

$$used(btarget)_\sigma^{I(3,t)} \leftrightarrow jbtaken_\sigma^{I(3,t)} \wedge il_\sigma^{I(3,t)} > \texttt{gf}$$

For the first signal, using the induction hypothesis (for the instruction register)

$$I^t_\pi = I_\sigma^{I(3,t)} \tag{112}$$

and correctness of the forwarding for A and B (in case they are used)

$$used(A)_\sigma^{I(3,t)} \;\rightarrow\; A(h^t_\pi) = A_\sigma^{I(3,t)} \tag{113}$$

$$used(B)_\sigma^{I(3,t)} \;\rightarrow\; B(h^t_\pi) = B_\sigma^{I(3,t)} \tag{114}$$

we argue as follows:

$$\begin{aligned} jbtaken^t_\pi &= jump^t_\pi \vee b^t_\pi \wedge bce^t_\pi.res \qquad \text{(construction)} \\ &= jump_\sigma^{I(3,t)} \vee b_\sigma^{I(3,t)} \wedge bce.res_\sigma^{I(3,t)} \qquad \text{(equations 112–114)} \\ &= jbtaken_\sigma^{I(3,t)} \qquad \text{(definition).} \end{aligned}$$

Moreover, here we rely on correctness of the instruction decoder and correct implementation of the switching function *res* of the branch condition evaluation (BCE) (see Sects. 2.1.3 and 2.1.4).

For the second signal (*btarget*), in case a jump or a branch instruction is executed

$$jbtaken^t_\pi = 1 = jbtaken_\sigma^{I(3,t)},$$

repeating the arguments above we derive:

$$\begin{aligned} btarget^t_\pi &= \begin{cases} pc^t_\pi +_{32} imm(h^t_\pi)[15]^{16} \circ imm(h^t_\pi) \circ 0_2 & b^t_\pi \wedge bce^t_\pi.res \\ A(h^t_\pi) & \text{otherwise} \end{cases} \qquad \text{(construction)} \\ &= \begin{cases} pc_\sigma^{I(3,t)} +_{32} imm_\sigma^{I(3,t)}[15]^{16} \circ imm_\sigma^{I(3,t)} \circ 0_2 & b_\sigma^{I(3,t)} \wedge bce.res_\sigma^{I(3,t)} \\ A_\sigma^{I(3,t)} & \text{otherwise} \end{cases} \qquad \text{(IH)} \\ &= btarget_\sigma^{I(3,t)} \qquad \text{(definition).} \end{aligned}$$

Let us also consider the update of the invisible registers, for instance register *ea*.4 (the effective address) in stage 4. Note, we have to show simulation of the invisible registers only if they are used. Thus, we argue

$$used(ea.4)_\sigma^{I(4,t+1)-1} \;\rightarrow\; used(ea.4.in)_\sigma^{I(4,t)} \;\rightarrow\; used(A)_\sigma^{I(4,t)}$$

and show

$$\begin{aligned} ea.4^t_\pi.in &= \begin{cases} A^t_\pi & cas.3^t_\pi \\ A^t_\pi +_{32} imm.3^t_\pi[15]^{16} \circ imm.3^t_\pi & \text{otherwise} \end{cases} \qquad \text{(construction)} \\ &= \begin{cases} A_\sigma^{I(4,t)} & cas_\sigma^{I(4,t)} \\ A_\sigma^{I(4,t)} +_{32} imm_\sigma^{I(4,t)}[15]^{16} \circ imm_\sigma^{I(4,t)} & \text{otherwise} \end{cases} \qquad \text{(lemma 79; IH)} \\ &= ea_\sigma^{I(4,t)} \qquad \text{(definition).} \end{aligned}$$

Therefore, for the *ea*.4 register we easily obtain:

$$ea.4^{t+1}_\pi = ea.4^t_\pi.in = ea_\sigma^{I(4,t)} = ea_\sigma^{I(4,t+1)-1}.$$

In case register R in stage k is not updated in cycle t ($/ue^t_k$), we clearly have

$$used(R.k)_\sigma^{I(k,t+1)-1} \;\rightarrow\; used(R.k)_\sigma^{I(k,t)-1}$$

and argue as follows:

$$\begin{aligned} R.k^{t+1}_\pi &= R.k^t_\pi \qquad \text{(construction)} \\ &= \begin{cases} R.k_\sigma^{I(k,t)} & vis(R.k) \\ R.k_\sigma^{I(k,t)-1} & \text{otherwise} \end{cases} \qquad \text{(IH)} \\ &= \begin{cases} R.k_\sigma^{I(k,t+1)} & vis(R.k) \\ R.k_\sigma^{I(k,t+1)-1} & \text{otherwise} \end{cases} \qquad \text{(lemma 80).} \end{aligned}$$

Internal Event Signals

First, for the external event vectors we show the following.

Lemma 152. *Assume $\hat{\Sigma}(t) < k$ where $k \leq 6$.*

$$eev_\sigma^{I(k,t)} = 0_2$$

Proof of lemma 152. Directly from the definitions we have

$$\mu(t) \leq \Sigma_{hard}(t) \leq \hat{\Sigma}(t).$$

From the above and assumptions, by definition of μ we conclude

$$live(k,t)$$

which by definition for some $t' \geq t$ implies

$$I(k,t) = I(6,t') \wedge ue_6^{t'}$$

and the claim follows.

$$\begin{aligned} eev_\sigma^{I(k,t)} &= eev_\sigma^{I(6,t')} \\ &= 0_2 \qquad \text{(equations 75, 76)} \end{aligned} \qquad \square$$

The latter result allows us to show correctness for the event signals collected in stages $k \in [1:5]$.

$$ue_k^t \;\rightarrow\; ev(k)^t = ev(k)_\sigma^{I(k,t)} \tag{115}$$

Proof of equation 115. The result can be easily derived from the statements below.

- $k = 1$. For the misalignment on fetch we show:

$$\begin{aligned} malf^t &\equiv ia_\pi^t[1:0] \neq 0_2 \qquad \text{(definition)} \\ &\equiv ia_\sigma^{I(1,t)}[1:0] \neq 0_2 \qquad \text{(IH)} \\ &\equiv malf_\sigma^{I(1,t)} \qquad \text{(definition).} \end{aligned}$$

  First, using the arguments above for walk w_I we derive the following.

$$used(w_I)_\sigma^{I(1,t)} \;\leftrightarrow\; /host(t) \wedge mca(1)^t[\texttt{>mf}] \tag{116}$$

  Proof of equation 116.

$$\begin{aligned} /host(t) \wedge mca(1)^t[\texttt{>mf}] &\leftrightarrow /host(t) \wedge /malf(t) \qquad \text{(definition)} \\ &\leftrightarrow /host_\sigma^{I(1,t)} \wedge /malf_\sigma^{I(1,t)} \qquad \text{(lemma 149.1)} \\ &\leftrightarrow /host_\sigma^{I(1,t)} \wedge il_\sigma^{I(1,t)} > 2 \qquad \text{(lemma 152)} \\ &\leftrightarrow used(w_I)_\sigma^{I(1,t)} \qquad \text{(definition)} \end{aligned} \qquad \square$$

  Having the result above, we argue for the page fault on fetch as follows:

$$\begin{aligned} pff^t &= /host(t) \wedge mca(1)^t[\texttt{>mf}] \wedge f(mmu_I^t.wout) \qquad \text{(definition)} \\ &= /host(t) \wedge mca(1)^t[\texttt{>mf}] \wedge match(trq_I{}_\sigma^{I(1,t)}, mmu_I^t.wout) \wedge f(mmu_I^t.wout) \qquad \text{(lemma 151.1)} \\ &= used(w_I)_\sigma^{I(1,t)} \wedge match(trq_I{}_\sigma^{I(1,t)}, mmu_I^t.wout) \wedge f(mmu_I^t.wout) \qquad \text{(equation 116)} \\ &= used(w_I)_\sigma^{I(1,t)} \wedge pfault(trq_I{}_\sigma^{I(1,t)}, mmu_I^t.wout) \qquad \text{(definition)} \\ &= used(w_I)_\sigma^{I(1,t)} \wedge pfault(trq_I{}_\sigma^{I(1,t)}, w_I{}_\sigma^{I(1,t)}) \qquad \text{(equation 73)} \\ &= pff_\sigma^{I(1,t)} \qquad \text{(definition).} \end{aligned}$$

  Note, the corresponding argument for the general-protection fault on fetch (gff) is analogous to the one presented above, and therefore is omitted.

- $k = 4$. For the misalignment on memory operation we show:

$$\begin{aligned} malm^t &\equiv mmask(4)^t[2:1] \wedge ea(4)^t[1:0] \neq 0_2 && \text{(definition)} \\ &\equiv mmask(4)_\sigma^{I(4,t)}[2:1] \wedge ea(4)_\sigma^{I(4,t)}[1:0] \neq 0_2 && \text{(IH)} \\ &\equiv mop(4)_\sigma^{I(4,t)} \wedge (d(4)_\sigma^{I(4,t)} \nmid \langle ea(4)_\sigma^{I(4,t)} \rangle) && \text{(lemma 8)} \\ &\equiv malm_\sigma^{I(4,t)} && \text{(definition).} \end{aligned}$$

- $k = 5$. Analogously to equation 116, using the arguments above for walk w_E we derive the following.

$$used(w_E)_\sigma^{I(5,t)} \;\leftrightarrow\; /host(t) \wedge mca(5)^t[\texttt{>mm}] \wedge mop.4_\pi^t \tag{117}$$

Having the result above, we argue for the page fault on memory operation as follows:

$$\begin{aligned} pfm^t &= /host(t) \wedge mca(5)^t[\texttt{>mm}] \wedge mop.4_\pi^t \wedge f(mmu_E^t.wout) && \text{(definition)} \\ &= /host(t) \wedge mca(5)^t[\texttt{>mm}] \wedge mop.4_\pi^t \wedge match(trq_{E\,\sigma}^{I(5,t)}, mmu_E^t.wout) \wedge f(mmu_E^t.wout) \\ &= used(w_E)_\sigma^{I(5,t)} \wedge match(trq_{E\,\sigma}^{I(5,t)}, mmu_E^t.wout) \wedge f(mmu_E^t.wout) && \text{(equation 117)} \\ &= used(w_E)_\sigma^{I(5,t)} \wedge pfault(trq_{E\,\sigma}^{I(5,t)}, mmu_E^t.wout) && \text{(definition)} \\ &= used(w_E)_\sigma^{I(5,t)} \wedge pfault(trq_{E\,\sigma}^{I(5,t)}, w_{E\,\sigma}^{I(5,t)}) && \text{(equation 74)} \\ &= pfm_\sigma^{I(5,t)} && \text{(definition).} \end{aligned}$$

The corresponding argument for the general-protection fault on memory operation (*gfm*) is omitted as well. □

Cause Pipeline Registers

Using the results above we infer correct update for registers of the cause pipeline. Thus, in case stage $k > 1$ is updated in cycle t, the cause register in stage k is written with the correct value.

$$\begin{aligned} ca.k_\pi^{t+1} &= ev(k)^t \vee ca.(k-1)_\pi^t && \text{(interconnect)} \\ &= ev(k)_\sigma^{I(k,t)} \vee ca.(k-1)_\sigma^{I(k,t)} && \text{(equation 115; IH)} \\ &= ca.k_\sigma^{I(k,t)} && \text{(definition)} \\ &= ca.k_\sigma^{I(k,t+1)-1} && \text{(lemma 80)} \end{aligned}$$

For the cause register in stage $k = 1$ the argument is analogous to the one presented above. Note, in case stage k is not updated in cycle t, correctness for the corresponding cause register is shown exactly as for the typical pipeline registers (see above).

Ghost Walk Registers

If pipeline stage $k = 1$ is updated in cycle t (ue_1^t), for the first ghost walk register of pipeline I we argue as follows.

$$\begin{aligned} w_I.1^{t+1} &= mmu_I^t.wout && \text{(interconnect)} \\ &= w_{I\,\sigma}^{I(1,t)} && \text{(equation 73)} \\ &= w_{I\,\sigma}^{I(1,t+1)-1} && \text{(definition)} \end{aligned}$$

For the first (and the only) ghost walk register of pipeline E we argue analogously.

$$\begin{aligned} w_E.5^{t+1} &= mmu_E^t.wout && \text{(interconnect)} \\ &= w_{E\,\sigma}^{I(5,t)} && \text{(equation 74)} \\ &= w_{E\,\sigma}^{I(5,t+1)-1} && \text{(lemma 80)} \end{aligned}$$

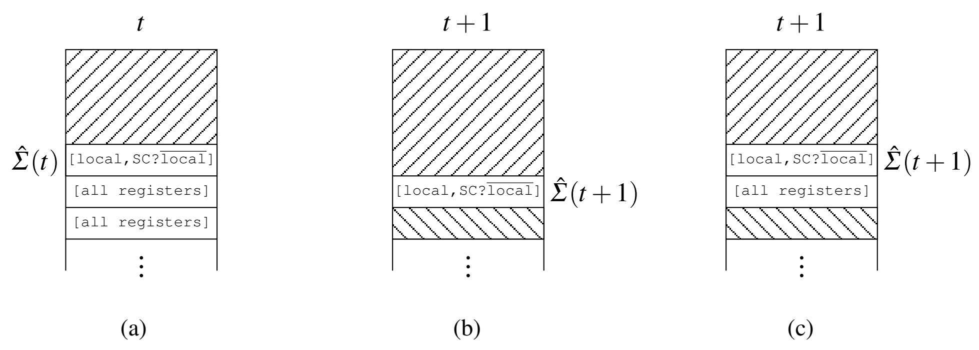


Fig. 42: Pipeline stages (not shaded) considered in the induction step (from t to $t+1$)

In case pipeline stage $k > 1$ is updated in cycle t (ue_k^t), the ghost walk register in stage k of pipeline I is of course written with the value of the corresponding register in the stage above.

$$\begin{aligned} w_I.k^{t+1} &= w_I.(k-1)^t && \text{(interconnect)} \\ &= w_{I\ \sigma}^{I(k-1,t)-1} && \text{(induction hypothesis)} \\ &= w_{I\ \sigma}^{I(k,t+1)-1} && \text{(definition)} \end{aligned}$$

Execution of eret

In case a legal *eret* instruction reaches the memory stage and the stage is updated, all stages above the memory stage become empty in cycle $t+1$ (part (2) of lemma 77).

$$ue_6^t \wedge iret_5^t \rightarrow \forall j \in [1:5] : /rfull_j^{t+1}$$

The special purpose registers

$$sprx \in \{sr, mode, nmode\}$$

are restored from their "exception" counterparts:

$$sprx_\pi^{t+1} = esprx_\pi^t = esprx_\sigma^{I(6,t)} = sprx_\sigma^{I(6,t)+1} = sprx_\sigma^{I(6,t+1)}.$$

The remaining SPR registers do not change:

$$spr_\pi^{t+1}(x) = spr_\pi^t(x) = spr_\sigma^{I(6,t)} = spr_\sigma^{I(6,t)+1} = spr_\sigma^{I(6,t+1)}.$$

None of the invisible registers in stage 6 are used except for the control bits, which are used always. For the control bits we derive:

$$R.6_\pi^{t+1} = R.5_\pi^t = R_\sigma^{I(6,t)} = R_\sigma^{I(6,t+1)-1}.$$

9.6.4 Speculative Execution

In this section we consider two cases of speculative execution, i.e., an execution with non-zero value of $\hat{\Sigma}$. First we consider the case in which the value of $\hat{\Sigma}$ increases, and then — the case in which the non-zero value of $\hat{\Sigma}$ stays unchanged. In both cases, using lemma 133 we obviously conclude the following.

$$k > \hat{\Sigma}(t) \rightarrow /rbr_k^t \tag{118}$$

In terms of correctness arguments nothing changes for stages

$$k > \hat{\Sigma}(t) + 1$$

in both cases, i.e., the same arguments as above apply for the speculative execution. The latter stages operate only with the visible registers from stages below $\hat{\Sigma}(t)+1$, and invisible registers from stages below $\hat{\Sigma}(t)$. As depicted in Fig. 42(a), in cycle t registers in these stages are simulated correctly. Moreover, since there is nothing to show for stages $k < \hat{\Sigma}(t+1)$, in the induction step below we argue only about stages

$$k \in [\hat{\Sigma}(t+1) : \hat{\Sigma}(t)+1].$$

In order to justify the case split on page 264, in the end of this section we argue that in the memory stage the value of $\hat{\Sigma}$ eventually resets to zero (equation 124).

$\hat{\Sigma}$ *Increases*

In case the value of $\hat{\Sigma}$ increases in cycle t

$$\hat{\Sigma}(t+1) = \hat{\Sigma}(t)+1$$

we have to argue only about the registers in stage

$$\begin{aligned} k &\in [\hat{\Sigma}(t+1) : \hat{\Sigma}(t)+1] \\ &\in \{\hat{\Sigma}(t+1)\}. \end{aligned}$$

According to lemma 134, the stage below $\hat{\Sigma}(t)$ is updated in cycle t.

$$ue^t_{\hat{\Sigma}(t)+1} \tag{119}$$

As depicted in Fig. 42(b), in the induction step we must consider i) the local invisible registers in stage k and ii) the non-local invisible registers in stage k, but only in case

$$\Sigma_{soft}(t+1) < \Sigma_{hard}(t+1).$$

For the first part we require the correct simulation of

- the visible registers in stage k and
- the local invisible registers in stage $k-1$

in cycle t, which we have according to Fig. 42(a). For the second part in addition to the above we require the correct simulation of

- the non-local invisible registers in stage $k-1$

in cycle t. According to Fig. 42(a) the latter registers are simulated correctly in cycle t in case the following holds.

$$\Sigma_{soft}(t) < \Sigma_{hard}(t) \tag{120}$$

Proof of equation 120. By contradiction; assume

$$\Sigma_{soft}(t) \geq \Sigma_{hard}(t)$$

and

$$\Sigma_{soft}(t+1) < \Sigma_{hard}(t+1).$$

The contradiction easily follows; using equations 119 and 88 we derive:

$$\begin{aligned} \Sigma_{soft}(t+1) &= \Sigma_{soft}(t)+1 && \text{(lemma 121.2)} \\ &= \hat{\Sigma}(t+1) && \text{(assumption)} \\ &\geq \Sigma_{hard}(t+1) && \text{(definition).} \end{aligned}$$ □

Therefore, we argue for the invisible registers in stage k as for the typical pipeline registers in Sect. 9.6.3 to show

$$used(R.k)^{I(k,t+1)-1}_{\sigma} \rightarrow R.k^{t+1} = R^{I(k,t+1)-1}_{\sigma}.$$

$\hat{\Sigma}$ *Remains Unchanged*

In case the value of $\hat{\Sigma}$ does not change in cycle t

$$\hat{\Sigma}(t+1) = \hat{\Sigma}(t)$$

we have to argue about the registers in stages

$$\begin{aligned} k &\in [\hat{\Sigma}(t+1) : \hat{\Sigma}(t)+1] \\ &\in [\hat{\Sigma}(t+1) : \hat{\Sigma}(t+1)+1]. \end{aligned}$$

According to lemma 134, the stage below $\hat{\Sigma}(t)$ is not updated in cycle t.

$$/ue^t_{\hat{\Sigma}(t)+1} \tag{121}$$

Moreover, since by definition of $\hat{\Sigma}$ we have

$$rfull^{t+1}_{\hat{\Sigma}(t)}$$

by part (3) of lemma 75 we conclude that stage $\hat{\Sigma}(t)$ is not updated either in cycle t.

$$/ue^t_{\hat{\Sigma}(t)} \tag{122}$$

As depicted in Fig. 42(c), in the induction step we must consider i) all registers in the stage below $\hat{\Sigma}(t)$, ii) the local invisible registers in stage $\hat{\Sigma}(t)$, and iii) the non-local invisible registers in stage $\hat{\Sigma}(t)$, but only if

$$\Sigma_{soft}(t+1) < \Sigma_{hard}(t+1).$$

For the first part, having equation 122, we require the correct simulation of

- all registers in stage $\hat{\Sigma}(t)+1$

in cycle t, which we have according to Fig. 42(a). For the second part, having equation 121, we require the correct simulation of

- the local invisible registers in stage $\hat{\Sigma}(t)$

in cycle t, which we have, again according to Fig. 42(a). Finally, for the third part, again using equation 121, we require the correct simulation of

- the non-local invisible registers in stage $\hat{\Sigma}(t)$

in cycle t. Analogous to the arguments above (Fig. 42(a)), the latter registers are simulated correctly in cycle t if the following holds.

$$\Sigma_{soft}(t) < \Sigma_{hard}(t) \tag{123}$$

Proof of equation 123. By contradiction; assume

$$\Sigma_{soft}(t) \geq \Sigma_{hard}(t)$$

and

$$\Sigma_{soft}(t+1) < \Sigma_{hard}(t+1).$$

The contradiction follows; using equations 118 and 121 we derive:

$$\begin{aligned} \Sigma_{soft}(t+1) &= \Sigma_{soft}(t) && \text{(lemma 121.1)} \\ &= \hat{\Sigma}(t+1) && \text{(assumption)} \\ &\geq \Sigma_{hard}(t+1) && \text{(definition).} \end{aligned}$$

□

Therefore, for visible registers in stage $\hat{\Sigma}(t)$ there is nothing to show. For the invisible registers in stage $\hat{\Sigma}(t)$ directly from the induction hypothesis we obtain

$$used(R.\hat{\Sigma}(t))_{\sigma}^{I(\hat{\Sigma}(t),t+1)-1} \rightarrow R.\hat{\Sigma}(t)^{t+1} = R_{\sigma}^{I(\hat{\Sigma}(t),t+1)-1}.$$

For stage $\hat{\Sigma}(t)+1$ we consider the following three cases.

- the register stage is stalled.
$$stall_{\hat{\Sigma}(t)+2}^{t}$$
We assume the stage has a real full bit. Then the real full bit is preserved.
$$rfull_{\hat{\Sigma}(t)+1}^{t+1}$$
Both visible and invisible registers in stage $k = \hat{\Sigma}(t)+1$ are not updated; their simulation holds by induction:
$$used(R.k)_{\sigma}^{I(k,t+1)-1} \rightarrow used(R.k)_{\sigma}^{I(k,t)-1}$$
and
$$\begin{aligned} R.k^{t+1} &= R.k^{t} && \text{(interconnect)} \\ &= \begin{cases} R_{\sigma}^{I(k,t)} & vis(R) \\ R_{\sigma}^{I(k,t)-1} & \text{otherwise} \end{cases} && \text{(IH)} \\ &= \begin{cases} R_{\sigma}^{I(k,t+1)} & vis(R) \\ R_{\sigma}^{I(k,t+1)-1} & \text{otherwise} \end{cases} && \text{(lemma 80).} \end{aligned}$$
- the register stage is emptied.
$$/ue_{\hat{\Sigma}(t)+1}^{t} \wedge ue_{\hat{\Sigma}(t)+2}^{t}$$
The visible registers are not updated; the simulation holds by induction as above. For the real full bit of stage $\hat{\Sigma}(t)+1$ in cycle $t+1$ using part (6) of lemma 75 we conclude:
$$/rfull_{\hat{\Sigma}(t)+1}^{t+1}.$$
Thus, there is nothing to show for the invisible registers.
- the register stage remains empty.
$$/rfull_{\hat{\Sigma}(t)+1}^{t} \wedge /rfull_{\hat{\Sigma}(t)+1}^{t+1}$$
Again, the visible registers are not updated and for the invisible registers there is nothing to show.

Repeated Rollback

Below we consider a special case, in which the rollback of stage 2 is performed *without* the reset of $\hat{\Sigma}$, i.e., without the update of the stage below $\hat{\Sigma}$. In this case we assume $pcres(t)$ and

$$\vec{\Sigma}(t) \in \{(0,2),(1,2)\}.$$

From the assumption and lemma 133 we have

$$rbr_{2}^{t} \wedge /rbr_{5}^{t} \rightarrow R(t) = 2.$$

The scheduling functions for stages 1 and 2 are rolled-back:

$$I(1,t+1) = I(2,t+1) = I(R(t),t) = I(2,t) = I(3,t)+1.$$

Note, the value of speculation stage $\vec{\Sigma}$ becomes

$$\vec{\Sigma}(t+1) = (0,2)$$

and the rollback is repeated in the next cycle, until the stage below $\hat{\Sigma}$ is eventually updated.

$\hat{\Sigma}$ Becomes Zero

Since after reset all pipeline stages are empty, by definition we have

$$\hat{\Sigma}(0) = 0.$$

According to lemma 134, the value of $\hat{\Sigma}$ increases only in cycles t in which the stage directly below $\hat{\Sigma}(t)$ is updated.

$$ue^t_{\hat{\Sigma}(t)+1}$$

Below we show that the value of $\hat{\Sigma}$ drops to zero after it reaches 5.

$$\hat{\Sigma}(t) = 5 \wedge ue^t_6 \;\rightarrow\; \hat{\Sigma}(t+1) = 0 \tag{124}$$

Proof of equation 124. Using equation 88, from the definition of $\hat{\Sigma}$ we derive

$$\hat{\Sigma}(t) = \Sigma_{hard}(t).$$

From lemma 130 we have that the rollback request signal in stage 5 is active in cycle t

$$rbr^t_5 = 1$$

and the claim follows by lemma 134. □

9.6.5 Exception Return

In this section we cover the details of restoring the program counters from the corresponding exception registers in the pipelined processor. In this case, according to lemma 134, we assume

$$\vec{\Sigma}(t) \in \{(0,2),(1,2)\}$$

and the active rollback request signal in stage $\hat{\Sigma}(t)$ in cycle t. Using the definition of the *misspec* signals we conclude:

$$\begin{aligned} rbr^t_{\hat{\Sigma}(t)} &\rightarrow rbr^t_2 \wedge /rbr^t_5 && \text{(lemma 133)} \\ &\rightarrow pcres(t) && \text{(interconnect)} \\ &\rightarrow rfull^t_2 \wedge iret^t_2 && \text{(definition).} \end{aligned}$$

Moreover, we assume that the stage below $\hat{\Sigma}(t)$ is updated in cycle t ($ue^t_{\hat{\Sigma}(t)+1}$), since otherwise the *pcres* signal stays active and the rollback is repeated in cycle $t+1$ (see Sect. 9.6.4, *Repeated Rollback*). Finally, in cycle $t+1$ we have

$$\hat{\Sigma}(t+1) = 0.$$

For the forwarded exception PCs (see Sect. 8.1.6), we argue about the forwarding circuits in the usual way as follows:

$$\begin{aligned} expc_F{}^t &= \begin{cases} C.4^t_\pi.in & top_{expc}[3]^t \\ C.k^t_\pi & top_{expc}[k]^t \wedge k > 3 \\ expc^t_\pi & \text{otherwise} \end{cases} && \text{(construction)} \\ &= \begin{cases} C^{I(k,t)}_\sigma & \min\{j \mid movg2s^{I(j,t)}_\sigma \wedge (xad^{I(j,t)}_\sigma = \texttt{spr[}expc\texttt{]})\} = k \in [4:6] \\ expc^{I(6,t)}_\sigma & \text{otherwise} \end{cases} && \text{(IH)} \\ &= expc^{I(3,t)}_\sigma && \text{(specification).} \end{aligned}$$

Using the arguments above, for the visible registers in stage 3 (program counters) we easily derive:

$$xpc^{t+1}_\pi = expc_F{}^t = expc^{I(3,t)}_\sigma = xpc^{I(3,t)+1}_\sigma = xpc^{I(3,t+1)}_\sigma.$$

For the remaining pipeline registers the arguments are analogous to those we already presented above in this section, and therefore are omitted. In particular, for stage 3 we obtain

$$rfull_3^{t+1} \wedge iret_3^{t+1}$$

and therefore for stages $k < 3$ there is nothing to show, since by part (1) of lemma 77 we have

$$\forall j \in [1:2] : /rfull_j^{t+1}.$$

In cycles $t' \geq t$ pipeline stages $j \leq k$ behind the legal *eret* instruction are drained; execution of the legal *eret* instruction finishes in the memory stage with update of the special purpose registers (see Sect. 9.6.3, *Execution of eret*).

9.6.6 Interrupt

Finally, we consider jumps to the interrupt service routine. In this case from the assumptions we conclude $jisr(t)$ and

$$\hat{\Sigma}(t) = 5.$$

From the definition of $\hat{\Sigma}$ using lemma 79 we derive:

$$rfull_5^t \wedge jisr_\sigma^{I(5,t)-1} \rightarrow jisr_\sigma^{I(6,t)}.$$

Since after the *jisr* in cycle t all real full bits except the last one ($k = 7$) are cleared, there is nothing to show for the invisible registers in stages ($k < 7$) in cycle $t + 1$. For the program counters we argue:

$$\begin{aligned}
(pc,dpc,ddpc)_\pi^{t+1} &= (8_4,4_4,0_4) && \text{(interconnect)}\\
&= (pc,dpc,ddpc)_\sigma^{I(6,t)+1} && \text{(specification)}\\
&= (pc,dpc,ddpc)_\sigma^{I(5,t)} && \text{(lemma 79)}\\
&= (pc,dpc,ddpc)_\sigma^{I(3,t+1)} && \text{(definition).}
\end{aligned}$$

For the exception cause register we have

$$\begin{aligned}
eca_\pi^{t+1} &= 0_{21} \circ f1(mca(6)^t) && \text{(interconnect)}\\
&= 0_{21} \circ f1(mca(6)_\sigma^{I(6,t)}) && \text{(lemma 125.1)}\\
&= eca_\sigma^{I(6,t)+1} && \text{(specification)}\\
&= eca_\sigma^{I(6,t+1)} && \text{(lemma 80),}
\end{aligned}$$

whereas for the exception data register we have

$$\begin{aligned}
edata_\pi^{t+1} &= ea.5_\pi^t \wedge (mca(6)^t[4:0] = 0_5) && \text{(interconnect)}\\
&= ea_\sigma^{I(6,t)} \wedge (mca(6)_\sigma^{I(6,t)}[4:0] = 0_5) && \text{(lemma 125.1)}\\
&= ea_\sigma^{I(6,t)} \wedge (il_\sigma^{I(6,t)} > 4) && \text{(definition)}\\
&= edata_\sigma^{I(6,t)+1} && \text{(specification)}\\
&= edata_\sigma^{I(6,t+1)} && \text{(lemma 80).}
\end{aligned}$$

For exception register *esprx* of SPR register

$$sprx \in \{sr, mode, nmode\}$$

we analogously obtain from the induction hypothesis:

$$esprx_\pi^{t+1} = sprx_\pi^t = sprx_\sigma^{I(6,t)} = esprx_\sigma^{I(6,t)+1} = esprx_\sigma^{I(6,t+1)}.$$

For the nested mode register we argue as follows.

$$\begin{aligned}
nmode_{\pi}^{t+1}[0] &= \begin{cases} 0 & user(q,t) \vee /icpt(q,t) \\ nmode_{\pi}^{t}[0] & \text{otherwise} \end{cases} && \text{(interconnect)} \\
&= \begin{cases} 0 & user_{\sigma}^{I(6,t)} \vee /icpt_{\sigma}^{I(6,t)} \\ nmode_{\sigma}^{I(6,t)}[0] & \text{otherwise} \end{cases} && \text{(IH)} \\
&= nmode_{\sigma}^{I(6,t)+1}[0] && \text{(specification)} \\
&= nmode_{\sigma}^{I(6,t+1)}[0] && \text{(lemma 80)}
\end{aligned}$$

The corresponding argument for the mode register is analogous to the one above, and therefore is omitted. For the remaining SPR registers we show:

$$\begin{aligned}
spr_{\pi}^{t+1}(x) &= \begin{cases} 0_{32} & x = 0_5 \\ spr_{\pi}^{t}(x) & \text{otherwise} \end{cases} && \text{(interconnect)} \\
&= \begin{cases} 0_{32} & x = 0_5 \\ spr_{\sigma}^{I(6,t)}(x) & \text{otherwise} \end{cases} && \text{(IH)} \\
&= spr_{\sigma}^{I(6,t)+1}(x) && \text{(specification)} \\
&= spr_{\sigma}^{I(6,t+1)}(x) && \text{(lemma 80).}
\end{aligned}$$

Note, moves to the SPR have no effect on interrupts, since move instructions cannot be interrupted by *continue* type interrupts. According to Table 9, the only instructions which can be interrupted by *continue* type interrupts are the arithmetic operations (*addi*, *add*, and *sub*) and the system call.

9.7 Correctness for TLBs

Reusing the results of Chap. 5 we argue that the translations in mmu_Y of processor q are contained in the TLB component of general computation $\tilde{c}_Y^q$.

$$\exists \tilde{c}_Y^{q,0} \forall t : sim_{tlb}(mmu_Y^{q,t}, \tilde{c}_Y^{q,t}) \tag{125}$$

This is lemma 36 from Chap. 5. Recall, the general stepping function ($\tilde{s}_Y$) for $\tilde{c}_Y$ was defined in the latter chapter. We adjust the definition of $\tilde{s}_Y$ in the obvious way to fit the hardware descriptions of the *multi-core* machine.

$$tadd_{Y\,X}^{q,t} \rightarrow \tilde{s}_Y^q(t) = \begin{cases} (winit, upa_X(mmu_Y^{q,t}), mmu_Y^{q,t}.pto_X) & winit_X(mmu_Y^{q,t}) \\ (wext, mmu_Y^{q,t}.w_X, pte_X(mmu_Y^{q,t})) & \text{otherwise} \end{cases}$$

$$tdrop_Y^{q,t} \rightarrow \tilde{s}_Y^q(t) = \begin{cases} (drop, mmu_Y^{q,t}.inva) & mmu_Y^{q,t}.invlpg \\ (drop, mmu_Y^{q,t}.inva.vm) & mmu_Y^{q,t}.vmflush \\ (drop, all) & \text{otherwise} \end{cases}$$

Analogous to Chap. 7, in order to incorporate the results of Chap. 5 (equation 125), it suffices to show the following.

$$\forall q : \textstyle\bigwedge_Y sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^{q,t+1}, mc^{NS(t+1)}.p(q))$$

Of course split cases on whether translations are added (see Sect. 9.7.1) or dropped (see Sect. 9.7.2) by the MMUs performing steps in cycle t (if any). Note, in the absence of queries to mmu_Y^q in cycle t, the simulation of TLB between the general and multi-core computation is maintained trivially on processor q in cycle $t+1$.

9.7.1 Adding Translations

In case step n is generated by mmu_Y on processor q in cycle t

$$s(n).(g,u) = (mmu_Y, q) \wedge n \in CS(t)$$

we show that the simulation of TLB between the general and multi-core computation is maintained for processor q in cycle $t+1$.

$$sim^{\text{ISA}}_{tlb}(\tilde{c}^{q,t+1}_Y, mc^{NS(t+1)}.p(q))$$

An analogous argument was made in Chap. 7. There we proved a corresponding result for the sequential machine (see Sect. 7.4.1). The latter proof can be used as it is for the multi-core machine; it suffices to reestablish the following arguments:

- for the page table origins and ASIDs

$$\begin{aligned} winit^{q,t}_{Y\,G} &\rightarrow & (pto, vmid)(q,t) &= (pto, vmid)(mc^n.p(q)) \\ winit^{q,t}_{Y\,U} &\rightarrow & (pto, npto, asid)(q,t) &= (pto, npto, asid)(mc^n.p(q)) \end{aligned}$$

- for the page table entry address

$$q \in TA_Y(t) \;\rightarrow\; ptea(mmu^{q,t}_Y) = ptea(s(n).o)$$

- for the page table entry

$$q \in TA_Y(t) \;\rightarrow\; pte(mmu^{q,t}_Y) = pte(mc^n.p(q), s(n).o)$$

Page Table Origins

First, we argue that the special purpose registers which are used for the address translation in cycle t do not change values in cycle t. These registers can be written only by the $movg2s$ instructions, which are illegal at the corresponding privilege levels.

$$guest(q,t) \rightarrow \quad mc^n.p(q).(pto, vmid) = mc^{NS(t)}.p(q).(pto, vmid) \tag{126}$$

$$user(q,t) \rightarrow mc^n.p(q).(pto, npto, asid) = mc^{NS(t)}.p(q).(pto, npto, asid) \tag{127}$$

Having that we obtain the first part directly from the induction hypothesis:

$$\begin{aligned} (pto, vmid)(q,t) &= (pto, vmid)^{q,I(6,q,t)}_{\sigma} && \text{(IH)} \\ &= (pto, vmid)^{q,ic(q,NS(t))}_{\sigma} && \text{(lemma 116)} \\ &= (pto, vmid)(mc^{NS(t)}.p(q)) && \text{(lemma 115)} \\ &= (pto, vmid)(mc^n.p(q)) && \text{(equations 126, 127).} \end{aligned}$$

We skip the second part; the omitted argument is literally the same as above.

Page Table Entry Address

Recall, correctness of the page table entry address read by mmu_Y in cycle t was established previously in Sect. 9.5.1 (see lemma 142).

Page Table Entry

For mmu_Y, using correctness of the translation cache output (established in Sect. 9.5.3), we obtain:

$$\begin{aligned} pte(mmu^{q,t}_Y) &= \begin{cases} pdout(4q + 2\mathbb{1}_{\{Y=E\}})^t_H & ptea(mmu^{q,t}_Y)[2] \\ pdout(4q + 2\mathbb{1}_{\{Y=E\}})^t_L & \text{otherwise} \end{cases} && \text{(interconnect)} \\ &= \begin{cases} tmout(n)_H & ptea(s(n).o)[2] \\ tmout(n)_L & \text{otherwise} \end{cases} && \text{(lemma 148)} \\ &= pte(mc^n, s(n).o) && \text{(lemma 14).} \end{aligned}$$

9.7.2 Dropping Translations

For step n in which processor q executes an invalidating instruction in cycle t

$$s(n).(t,u) = (core,q) \wedge (invlpg_\sigma \vee flusht_\sigma)^{q,ic(q,n)} \wedge n \in CS(t) \wedge exec(q,t)$$

we show (as above) that the simulation of TLB between the general and multi-core computation is maintained for processor q in cycle $t+1$.

$$\bigwedge_Y sim_{tlb}^{\text{ISA}}(\tilde{c}_Y^{q,t+1}, mc^{NS(t+1)}.p(q))$$

Again, a similar result was proven for the sequential machine in Chap. 7 and, again, we can reuse the proof. For the multi-core machine we need to update the following arguments:

- for the MMU control signals

$$\begin{aligned} mmu_Y^{q,t}.invlpg &\leftrightarrow invlpg_\sigma^{q,i} \wedge \overline{user_\sigma^{q,i}} \\ mmu_Y^{q,t}.vmflush &\leftrightarrow flusht_\sigma^{q,i} \wedge guest_\sigma^{q,i} \\ mmu_Y^{q,t}.flush &\leftrightarrow flusht_\sigma^{q,i} \wedge host_\sigma^{q,i} \end{aligned}$$

- for the invalidation address

$$mmu_Y^{q,t}.inva = inva_\sigma^{q,i}$$

where for convenience we abbreviate the instruction count on processor q in cycle t:

$$i = ic(q,n) = ic(q,NS(t)) = I(q,6,t).$$

We argue only about one of the three invalidation signals above, namely for $vmflush$.

$$\begin{aligned} mmu_Y^{q,t}.vmflush &= flusht.5_\pi^{q,t} \wedge guest(q,t) && \text{(interconnect)} \\ &= flusht_\sigma^{q,i} \wedge guest_\sigma^{q,i} && \text{(IH)} \end{aligned}$$

Arguments for the remaining control signals are identical. Finally, for the invalidation address we argue in the same spirit and obtain:

$$\begin{aligned} mmu_Y^{q,t}.inva &= \begin{cases} vmid(q,t) \circ C.5_\pi^{q,t}[27:0] & guest(q,t) \\ C.5_\pi^{q,t} & \text{otherwise} \end{cases} && \text{(interconnect)} \\ &= \begin{cases} vmid_\sigma^{q,i} \circ C_\sigma^{q,i}[27:0] & guest_\sigma^{q,i} \\ C_\sigma^{q,i} & \text{otherwise} \end{cases} && \text{(IH)} \\ &= inva_\sigma^{q,i} && \text{(definition).} \end{aligned}$$

9.8 Maintaining Invariants

In this section we assume that the pipeline of processor q executes at the privilege level of user or guest, i.e., the address translation is used. For convenience, we repeat the statements of invariant 10

$$\hat{\Sigma}(q,t+1) \le k \wedge rfull_k^{q,t+1} \to w_I.k^{q,t+1} \in \begin{cases} mc^{NS(t+1)}.p(q).tlb^\circ & user(q,t+1) \\ mc^{NS(t+1)}.p(q).tlb & guest(q,t+1) \end{cases}$$

$$\vec{\Sigma}(q,t+1) \le (4,5) \wedge rfull_5^{q,t+1} \to w_E.5^{q,t+1} \in \begin{cases} mc^{NS(t+1)}.p(q).tlb^\circ & user(q,t+1) \\ mc^{NS(t+1)}.p(q).tlb & guest(q,t+1) \end{cases}$$

and invariant 11, formulated for cycle $t+1$ in which the address translation is used.

$$\begin{aligned} \hat{\Sigma}(q,t+1) \le k \wedge I(q,k,t+1) = i \wedge rfull_k^{q,t+1} &\to match(trq_{I\,\sigma}^{q,i-1}, w_I.k^{q,t+1}) \\ \vec{\Sigma}(q,t+1) \le (4,5) \wedge I(q,5,t+1) = i \wedge rfull_5^{q,t+1} &\to match(trq_{E\,\sigma}^{q,i-1}, w_E.5^{q,t+1}) \end{aligned}$$

Recall, in Sect. 9.2.4 we showed that both invariants (10 and 11) hold after the reset, in cycle $t = 0$.

9.8.1 Translations in Use and Invalidation of TLB

To show that invariants 10 and 11 are maintained in cycle $t+1$, we split cases on stage k of the ghost walk pipeline of processor q and cover the input stages first. Thus, for mmu_I^q we consider stage $k=1$ of processor q in cycles t in which the stage is updated ($ue_1^{q,t}$). We obtain:

$$\begin{aligned} w_I.1^{q,t+1} &= mmu_I^{q,t}.wout \qquad \text{(interconnect)} \\ &\in \begin{cases} tlb_U(mmu_I^{q,t}) & user(q,t) \\ tlb_G(mmu_I^{q,t}) & guest(q,t) \end{cases} \qquad \text{(interconnect; spec.)} \\ &\subseteq \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases} \qquad \text{(IH)}. \end{aligned}$$

For mmu_E we consider stage $k=5$ with the active update enable signal (ue_5^t). Thus, given that a memory operation is executed ($mop.4_{\pi}^{q,t}$), we analogously obtain:

$$\begin{aligned} w_E.5^{q,t+1} &= mmu_E^{q,t}.wout \qquad \text{(interconnect)} \\ &\in \begin{cases} tlb_U(mmu_E^{q,t}) & user(q,t) \\ tlb_G(mmu_E^{q,t}) & guest(q,t) \end{cases} \qquad \text{(interconnect; spec.)} \\ &\subseteq \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases} \qquad \text{(IH)}. \end{aligned}$$

In both cases we argue as follows. Since the corresponding stage k is updated, there is no invalidating instruction in the memory stage. Using definitions from Sect. 8.1.5 we derive:

$$\begin{aligned} ue_k^t \rightarrow /haz_k^t &\rightarrow /inval(mmu_Y^{q,t}) \qquad \text{(interconnect, p. 184)} \\ &\rightarrow /(rfull_5^{q,t} \wedge (invlpg.5^{q,t} \vee flusht.5^{q,t})) \qquad \text{(interconnect, p. 183)} \\ &\rightarrow /(invlpg_{\sigma} \vee flusht_{\sigma})^{q,I(q,6,t)} \qquad \text{(IH; lemma 79)} \\ &\rightarrow /(invlpg_{\sigma} \vee flusht_{\sigma})^{q,ic(q,NS(t))} \qquad \text{(lemma 116)}. \end{aligned}$$

The latter obviously yields:

$$mc^{NS(t)}.p(q).tlb \subseteq mc^{NS(t+1)}.p(q).tlb.$$

In cycle $t+1$ for the ghost walk registers in stage $k>1$ of pipeline I on processor q we argue directly from the induction hypothesis:

$$\begin{aligned} w_I.k^{q,t+1} &= \begin{cases} w_I.(k-1)^{q,t} & ue_k^{q,t} \\ w_I.k^{q,t} & \text{otherwise} \end{cases} \qquad \text{(interconnect)} \\ &\subseteq \begin{cases} mc^{NS(t)}.p(q).tlb^{\circ} & user(q,t) \\ mc^{NS(t)}.p(q).tlb & guest(q,t) \end{cases} \qquad \text{(IH)}. \end{aligned}$$

The corresponding argument for the ghost walk registers of pipelines E is analogous, and therefore is omitted.

Finally, in case there is no invalidating instruction in the memory stage in cycle t we argue exactly as above to finish the induction step. Otherwise, we proceed to show that the translations used in the pipeline are never invalidated. We split cases on the execution mode of the pipeline. On the level of host the address translation is not used, and therefore there is nothing to show for translations in the ghost pipeline. On the levels of guest and user the address translation is used. The invalidation of the MMUs is performed only in cycles in which an invalidating instruction is *executed*.[2]

[2] Note, according to Table 9, invalidating instructions cannot be interrupted by *continue* type interrupts.

$$\begin{aligned} inval(mmu_Y^{q,t}) &\rightarrow exec(q,t) && \text{(interconnect)} \\ &\rightarrow /jisr(q,t) && \text{(specification)} \end{aligned}$$

We use that both, invalidation of any guest addresses on the level of guest, as well as invalidation of any addresses on the level of user are illegal:

$$\begin{aligned} /jisr(q,t) &\rightarrow \begin{cases} /inval(mmu_Y^{q,t}) & user(q,t) \\ mmu_Y^{q,t}.inva \notin A_G & guest(q,t) \end{cases} && \text{(construction)} \\ &\rightarrow \begin{cases} /(invlpg_\sigma \vee flusht_\sigma)^{q,I(q,6,t)} & user(q,t) \\ inva_\sigma^{q,I(q,6,t)} \notin A_G & guest(q,t) \end{cases} && \text{(IH).} \end{aligned}$$

At the level of user we immediately conclude as above

$$mc^{NS(t)}.p(q).tlb \subseteq mc^{NS(t+1)}.p(q).tlb$$

and we are done. At the level of guest, from the induction hypothesis we know that in cycle t the ghost walk register in stage k of pipeline Y matches translation request Y of the processor for execution of instruction at stage $k+1$ in cycle t. Thus, the walks in the ghost pipeline are the guest walks, and the guest walks are not invalidated at the level of guest as we argued above.

$$\begin{aligned} guest(q,t) \wedge match(trq_{Y\ \sigma}^{q,I(q,k,t)-1}, w_Y.k^{q,t}) &\rightarrow w_Y.k^{q,t}.upa \in A_G \\ &\rightarrow w_Y.k^{q,t} \in mc^{NS(t+1)}.p(q).tlb \end{aligned}$$

Summarizing the arguments above, we conclude (invariant 10) that in cycle $t+1$ the ghost walk registers in stage k of pipeline Y are contained in the specification/constructed TLB of ISA configuration $NS(t+1)$.

$$w_Y.k^{q,t+1} \in \begin{cases} mc^{NS(t+1)}.p(q).tlb^\circ & user(q,t) \\ mc^{NS(t+1)}.p(q).tlb & guest(q,t) \end{cases}$$

Therefore, it remains to show that the execution mode on processor q does not change in cycle t.

$$mode(q,t+1) = mode(q,t)$$

In case the mode changes in cycle t, by lemma 78 we immediately obtain

$$ue_6^{q,t} \wedge (jisr.5_\pi \vee eret.5_\pi)^{q,t}.$$

In both cases by lemma 76 there is nothing to show for registers in the ghost walk pipeline.

$$\forall k \in [1:5] : /rfull_k^{t+1}$$

9.8.2 Matching Translation Requests

Finally, we proceed to show (invariant 11) that in cycle $t+1$ the ghost walk register in stage k of pipeline Y on processor q matches translation request Y of instruction

$$i = I(q,k,t+1)$$

at stage $k+1$ of processor q in cycle $t+1$.

$$match(trq_{Y\ \sigma}^{q,i-1}, w_Y.k^{q,t+1})$$

As above, we split cases on stage k of the ghost walk pipeline of processor q, and again we cover the input stages first. For cycles t in which stage $k=1$ of processor q is updated ($ue_1^{q,t}$), using part (1) of lemma 151 we obtain:

$$\begin{aligned} match(trq_{I\ \sigma}^{q,I(q,1,t)}, mmu_I^{q,t}.wout) &\leftrightarrow match(trq_{I\ \sigma}^{q,I(q,1,t)}, w_I.1^{q,t+1}) && \text{(interconnect)} \\ &\leftrightarrow match(trq_{I\ \sigma}^{q,I(q,1,t+1)-1}, w_I.1^{q,t+1}) && \text{(definition).} \end{aligned}$$

For cycles t in which stage $k = 5$ of pipeline E on processor q is updated ($ue_5^{q,t}$), from part (2) of lemma 151 — given that a memory operation is executed ($mop.4_\pi^{q,t}$) — we analogously obtain:

$$\begin{aligned} match(trq_{E\ \sigma}^{q,I(q,5,t)}, mmu_E^{q,t}.wout) &\leftrightarrow match(trq_{E\ \sigma}^{q,I(q,5,t)}, w_E.5^{q,t+1}) && \text{(interconnect)} \\ &\leftrightarrow match(trq_{E\ \sigma}^{q,I(q,5,t+1)-1}, w_E.5^{q,t+1}) && \text{(lemma 80).} \end{aligned}$$

In case pipeline stage k of processor q is not updated in cycle t ($/ue_k^{q,t}$), we argue as above, simply using the induction hypothesis for stage k:

$$match(trq_{Y\ \sigma}^{q,I(q,k,t)-1}, w_Y.k^{q,t}) \ \leftrightarrow\ match(trq_{Y\ \sigma}^{q,I(q,k,t+1)-1}, w_Y.k^{q,t+1}).$$

If the ghost walk register in stage $k > 1$ of pipeline I on processor q is updated in cycle t ($ue_k^{q,t}$), we argue analogously, using the induction hypothesis for stage $k-1$:

$$match(trq_{Y\ \sigma}^{q,I(q,k-1,t)-1}, w_Y.(k-1)^{q,t}) \ \leftrightarrow\ match(trq_{Y\ \sigma}^{q,I(q,k,t+1)-1}, w_Y.k^{q,t+1}).$$

This ends the induction step and therefore the correctness proof for the pipelined multi-core MIPS machine with nested translation. The latter proof is the main result of this thesis.

Conclusion

Initially, the goals of this thesis were to i) strengthen the virtualization capabilities of the pipelined multi-core machine from [Pau16] and ii) prove correctness of the resulting design. For the first goal, aiming at the hardware support for hypervisors, we had to replace the address translation scheme used in [Pau16] with a more general one, permitting two phases of address translation. For the second goal, we had to investigate the correctness proofs presented in [Pau16] and, if necessary, repair the arguments broken by integration of the two-phase translation scheme.

Given the latter two goals, we briefly describe the main contributions of this thesis.

- First, we presented the formal specification of the nested address translation. Then, we proposed a simple, still not completely trivial, implementation of the MMU for the nested translation (nested MMU). The latter implementation was proven correct independent of any particular machine semantics. Namely, for any valid sequence of MMU queries we showed that computation of the nested MMU was simulated by a computation in the general semantics, formalized in this thesis. As a result, we separated correctness of implementation of the nested address translation from the overall machine correctness.
- In order to improve overall performance, we additionally introduced the address space identifiers (ASIDs) to designate entries in the processor-local translation cache, the translation look-aside buffer (TLB). The latter enabled the possibility to invalidate TLB entries belonging to particular processes, while keeping other entries intact. At the same time, in order to keep things simple, we dropped the access and dirty bits from our specification. To make the nested translation invisible for the guests, we also introduced a simple mechanism of intercepts.
- We demonstrated how to integrate the nested MMU and prove its correctness for various machines. For that we integrated the nested address translation semantics into the MIPS-86 ISA specification [Sch13a]. Then we considered both, the sequential and the pipelined implementations of the latter specification, resp. single-core and multi-core. For both implementations we obtained liveness and correctness in complete paper and pencil proofs. Within the proof we used certain modularity, which allowed us to prevent the arguments from repeating.

It is worth mentioning that in this thesis the proof for the pipelined machine used the following new auxiliary mathematical concepts.

- The first concept was introduced to keep track of instructions violating the software conditions, and formalized in the definition of *speculation stage* Σ_{soft}. This definition turned out to be crucial in the pipelined implementation to establish correctness for the output of the instruction cache, and therefore of the fetched instruction.
- The second concept was introduced to keep track of non-live instructions, i.e., instructions present in the pipeline but eventually discarded on rollbacks. The second concept was formalized in the definition of *lowest non-live circuit stage* μ; it turned out to be crucial to justify our “early” definition of the oracle inputs. Recall that in Sect. 9.2.5 we defined the

oracle inputs for instructions in stages below μ and in cycles in which these instructions progress down the pipeline.

In the next section we discuss the obtained results in more detail, from the perspective of problems we encountered in the process of working on this thesis. While discussing the latter problems, we also inform the reader on how these (or the counterpart) problems were tackled outside this thesis. In the end, we briefly discuss future work.

Discussion

Overall, the results presented in this thesis hinge heavily on both already published works [KMP14, PBLS16] and ongoing projects [LOP]. Some of the results presented here required us to repeat certain arguments from the aforementioned works; certain arguments from [Pau16] were expected to change simply due to a slightly more advanced machine design. In a machine with an additional level of execution, among the other things, we had to elaborated the following. Recall from Sect. 2.4 that one of the guard conditions restricting computations in [Sch13a] demands that the translations used for instruction execution are taken from a local cache for translations, i.e., the translation look-aside buffer (TLB). In [Pau16] translations used by instructions in the upper pipeline stages were never invalidated simply due to the following observations: i) invalidating instructions are illegal at the execution level which uses the address translation and ii) change of the execution level effectively drains the pipeline. In this thesis, with three levels of execution, the arguments above did not apply since invalidating instructions are legal at the intermediate level (of guest), which uses the address translation.

Development and integration of the two-phase translation scheme into the pipeline from [Pau16] turned out to be rather technical and required a moderate effort. However, in the process of elaborating the details in the correctness proof for the resulting design, we discovered a flaw in the original proof of [Pau16]. There sampling of the oracle inputs for the ISA computation was performed as usual in the memory stage, in cycles of execution of the corresponding instructions. At the same time, the main induction hypothesis about the content of the implementation registers, including those located in the upper pipeline stages, was formulated using the oracle inputs defined in future cycles (from perspective of the cycle considered in the induction hypothesis), with *no* assumptions that the latter cycles actually exist. Existence of these (future) cycles in which the machine executes instructions clearly follows from the machine liveness, which is normally shown *after* the machine correctness. After a few collective discussions, we decided to prove the machine liveness before correctness. In the presence of rollbacks, this approach allowed us to show uniqueness of cycles in which instructions that are not rolled-back *progress* down the pipeline. As a result, we sampled the oracle inputs for instructions that are not rolled-back exactly in their progress cycles. The resulting arguments ended up to be somewhat more lengthy than we expected. In [LOP], which is developed in parallel to this thesis, the correctness proof is performed only for instructions which are not rolled-back. Completed with the liveness proof afterwards, the latter approach is expected to be more concise.

In general, if it concerns major changes to the verified designs, we always proceed one step at a time. Initially, being unaware of the problems described above, we decided to allow *self-modifying code* in [Pau16] as one such step. Surprisingly, in the presence of self-modifying code, the correctness proof for the pipelined machine with address translation became considerably more difficult. Note that the presence of self-modifying code naturally forces us to introduce a software condition that

i) in single-core pipelined machines forbids to modify the words of instructions while the latter are executed in the pipeline, i.e., after fetch and before commit in the memory stage, and
ii) in multi-core pipelined machines forbids to modify the words of instructions while the latter are executed in the pipelines of the multi-core machine.

The difficulty arose for the following reason: in order to use that software condition we must first argue that the guard conditions are respected in the computations produced by our pipelined implementation. In turn, some of these guard conditions can be verified only provided that the software conditions hold. Luckily, this mutual dependence could be broken [Obe17] by splitting execution of instructions into phases. Thus, in Sect. 2.2.2 we split execution of instruction into two phases: the fetch and the execute. Recall, the fetch phase encompasses computations of all ISA signals necessary to fetch the current instruction, and ends once the instruction word was fetched. Computations remaining to complete execution of the current instruction constitute the execute phase. The sets of the software and guard conditions were split accordingly. This allowed us to strengthen the formulations involved, and thus resolve the problem of mutual dependence described above. The software conditions which apply in the fetch phase, like the software condition about self-modifying code, were expected to hold if the guard conditions were satisfied *before* the fetch. The latter assumption was reflected in our induction hypothesis (see Sect. 9.2.4).

In [LOP] the problem described above is outmaneuvered at level of ISA specification. There the model is extended to include *instruction buffers* — load buffers for instructions. In the latter model, instruction buffers are filled with data (instructions) non-deterministically, on dedicated ISA steps, and emptied on interrupts and returns from exceptions. Instruction execution is restricted only to the instructions found in the instruction buffer and, of course, matching the actual instruction address. This makes the new model of [LOP] non-deterministic also in the choice of instructions for execution. Thus, in the latter model instruction words from the instructions buffer which are outdated due to self-modifying code are allowed to be executed. The corresponding correctness proofs in [LOP] are expected to be more concise than their counterparts from this dissertation. Basically, allowing instruction buffers in [LOP] pushes the problems introduced by self-modifying code out of the pipeline correctness proof, where it boils down to (not so amusing) application of the software condition described above.

Another interesting observation was pointed out by Jonas Oberhauser. Essentially, instruction buffers and their related problems are quite similar to translation look-aside buffers (TLBs) and problems tackled in this thesis. Similar to instruction buffers, TLBs are filled with data (translations) non-deterministically, on dedicated ISA steps. In general, non-deterministic mechanisms leave a lot of space for concrete implementations, and as we discussed above, allow to streamline correctness proofs for pipelined designs. Moreover, the provided functionality, such as TLB invalidating instructions, usually allows to construct certain programming disciplines to obtain effectively deterministic programming models on the higher layers of the model stack. Reduction theorems are proven to justify abstraction of the mechanisms as above, provided that a certain, derived in the proof process software discipline is obeyed. Such disciplines might require some extra (usually minor) effort for programming in high-level languages, or (ideally) could be entirely integrated into compilers.

Future Work

First and foremost, we are interested in adding support for the inter-processor interrupts, which requires adding the advanced programmable interrupt controllers (APICs). Much work on this topic has already been completed in the process of working on this thesis. Thus, at the moment of writing we strongly believe that integration of APICs into the current design requires only minor local changes, and does not affect the structure of the arguments presented here or in [LOP].

Still, in order to integrate the results of this thesis (or of [LOP]) with the results from [Obe17], where the problem of store buffer reduction was tackled, we have to adjust the specification for external interrupts. Namely, we have to change model of [Sch13a] s.t. delivery of external interrupts can be delayed. The ability to postpone delivery of the inter-processor interrupts for arbitrary number of cycles is crucial to reach higher layers of the model stack, which provide semantics without store buffers. Using the new mechanisms we also should be able to postpone the activation of JISR to the subsequent instructions if an external interrupt occurs during the memory operation. Such change would allow us keep the usual return type (repeat) for external interrupts.

Finally, one would think of adding devices. Thus, at the moment of writing the hard drive has already been added as a sample device into the pipelined machine of [LOP]. The latter result has to be extended to meet the specifications from [Sch13a]. According to the latter specifications, the devices are interconnected with local APICs and I/O APIC into a single device sub-system, where all devices are connected and communicate via the interrupt bus. We believe that a moderate effort is required to finish the implementation of the device sub-system as specified in [Sch13a], and integrate the corresponding correctness arguments into the overall correctness proof.

References

ABCC66. R. Adair, R. Bayles, L. Comeau, and R. Creasy. A Virtual Machine System for the 360/40. Technical report, IBM Corp., Cambridge Scientific Center, May 1966.

Age09. Ole Agesen. Software and Hardware Techniques for x86 Virtualization. Technical paper, VMware, Inc., August 2009. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/software_hardware_tech_x86_virt.pdf`.

APST10. E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. In *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, volume 6217 of *LNCS*, pages 71–85. Springer, 2010.

ARM14. ARM and QUALCOMM. Enabling the Next Mobile Computing Revolution with Highly Integrated ARMv8-A based SoCs. White paper, ARM Limited/QUALCOMM Technologies, Inc., July 2014. `https://www.arm.com/files/pdf/ARM_Qualcomm_White_paper_Final.pdf`.

BC13. Thomas Braibant and Adam Chlipala. Formal Verification of Hardware Synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 213–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Bev89. William R. Bevier. Kit and the Short Stack. *Journal of Automated Reasoning*, 5(4):519–530, December 1989.

Bey05. Sven Beyer. *Putting It All Together — Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, 2005.

BHMY89. William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An Approach to Systems Verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

BJK$^+$03. Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang J. Paul. Instantiating Uninterpreted Functional Units and Memory System: Functional Verification of the VAMP. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods, Proc. 12th IFIP WG 10.5 Advanced Research Working Conference (CHARME'03),* L'Aquila, Italy, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

CPS13. Ernie Cohen, Wolfgang Paul, and Sabine Schmaltz. Theory of Multi Core Hypervisor Verification. In Peter van Emde Boas, Frans C. A. Groen, Giuseppe F. Italiano, Jerzy Nawrocki, and Harald Sack, editors, *Proc. SOFSEM'13: Theory and Practice of Computer Science,* Spindleruv Mlyn, Czech Republic, volume 7741 of *LNCS*, pages 1–27. Springer, 2013.

CVS$^+$17. Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A Platform for High-level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, August 2017.

DDB08. Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model Stack for the Pervasive Verification of a Microkernel-based Operating System. In Bernhard Beckert and Gerwin Klein, editors, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.

ELMC18. Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based Cryptojacking. *CoRR*, abs/1803.02887, 2018. `http://arxiv.org/abs/1803.02887`.

Hil05. Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Saarbrücken, 2005.

Int16. Intel Corporation. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual*. Intel Corp., September 2016. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf`.

Int18. Intel Corporation. Intel Analysis of Speculative Execution Side Channels. White paper, Intel Corp., January 2018. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/analysis-of-speculative-execution-side-channels-white-paper.pdf`.

KAK$^+$. Gerwin Klein, June Andronick, Ihor Kuz, Toby Murray, Gernot Heiser, and Matthew Fernandez. Formal Verification at Scale. *Communications of the ACM*. To appear.

KGG$^+$18. Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, January 2018.

KMP14. M. Kovalev, S.M. Müller, and W.J. Paul. *A Pipelined Multi-core MIPS Machine: Hardware Implementation and Correctness Proof*, volume 9000 of *LNCS*. Springer, 2014.

Kov13. Mikhail Kovalev. *TLB Virtualization in the Context of Hypervisor Verification*. PhD thesis, Saarland University, Saarbrücken, 2013.

Krö01. Daniel Kröning. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Saarbrücken, 2001.

LOP. P. Lutsyk, J. Oberhauser, and W.J. Paul. *A Pipelined Multi-core Machine with Operating Systems Support: Hardware Implementation and Correctness Proof.* LNCS. To appear.

LSG$^+$18. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

Lut14. Petro Lutsyk. Pipelined MIPS Processor with a Store Buffer. Master's thesis, Saarland University, Saarbrücken, 2014.

Mai14. Giorgi Maisuradze. Implementing and Debugging a Pipelined Multi-Core MIPS Machine. Master's thesis, Saarland University, Saarbrücken, 2014.

Meg12. Natarajan Meghanathan. Virtualization of Virtual Memory Address Space. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, pages 732–737, New York, NY, USA, 2012. ACM.

Moo03. J. Strother Moore. *A Grand Challenge Proposal for Formal Methods: A Verified Stack*, pages 161–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

MP00. Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

Obe17. Jonas Oberhauser. *Justifying The Strong Memory Semantics of Concurrent High-Level Programming Languages for System Programming*. PhD thesis, Saarland University, Saarbrücken, 2017.

Ols14. Loris Olsem. Development Toolchain for MIPS. Bachelor's thesis, Saarland University, Saarbrücken, 2014.

Pau16. W.J. Paul. Multi-Core System Architecture. Lecture notes, Saarland University, Saarbrücken, 2016. `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur/ws15/books/mcsysbook.pdf`.

PBLS16. W.J. Paul, C. Baumann, P. Lutsyk, and S. Schmaltz. *System Architecture: An Ordinary Engineering Discipline*. Springer, 2016.

RIS18. RISC-V Foundation. RISC-V ISA, 2018. `https://riscv.org/risc-v-isa/`.

Sch13a. Sabine Schmaltz. MIPS-86 — A Multi-Core MIPS ISA Specification. Technical report, Saarland University, Saarbrücken, 2013. `http://www-wjp.cs.uni-saarland.de/publikationen/SchmaltzMIPS.pdf`.

Sch13b. Sabine Schmaltz. *Towards the Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C*. PhD thesis, Saarland University, Saarbrücken, 2013.

Sch14a. Oliver Schmitt. Design and Verification of Memory Management Units for Single-Core CPUs. Bachelor's thesis, Saarland University, Saarbrücken, 2014.

Sch14b. Konstantin Schwarz. Integration of Inter-Processor Interrupts into Multi-Core Machines. Bachelor's thesis, Saarland University, Saarbrücken, 2014.

Sch16a. Felix Schmidt. Correctness of a Pipelined MIPS Machine with Interrupts and a Cache Memory System. Master's thesis, Saarland University, Saarbrücken, 2016.

Sch16b. Konstantin Schwarz. Correctness of Multi-Core Machines with Non-Pipelined Processors and Inter-Processor Interrupts. Master's thesis, Saarland University, Saarbrücken, 2016.

SK17. Hira Syeda and Gerwin Klein. Reasoning about Translation Lookaside Buffers. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 490–508, Maun, Botswana, May 2017.

VCAD15. Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. *Modular Deductive Verification of Multiprocessor Hardware Designs*, pages 109–127. Springer International Publishing, Cham, 2015.

Ver07. Verisoft Consortium. The Verisoft Project, 2003–2007. `http://www.verisoft.de/`.

Ver10. Verisoft XT Consortium. The Verisoft XT Project, 2007–2010. `http://www.verisoftxt.de/`.

Vij16. Muralidaran Vijayaraghavan. *Modular Verification of Hardware Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2016.

Vir18. libvirt Virtualization API. *Application Development. Description of the XML schemas for domains*. libvirt Project, 2018. `https://libvirt.org/formatdomain.html#elementsMemoryBacking`.

WLPA16. Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

WZB$^+$16. Andy Wright, Sizhuo Zhang, Thomas Bourgeat, Muralidaran Vijayaraghavan, Jamey Hicks, and Arvind. Riscy Processors: A collection of open-sourced RISC-V processors. In 4th RISC-V Workshop, July 2016.

Zah16. Shahd Zahran. Implementing and Debugging a Pipelined MIPS Machine with Interrupts and Multi-Level Address Translation. Master's thesis, Saarland University, Saarbrücken, 2016.

Index