Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU WIEN UB Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK Faculty of Informatics

Analysis of the Failure Behavior of Memory Management Units

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Oliver Hechinger, BSc
Matrikelnummer 0726503

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger

Wien, 03.12.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Oliver Hechinger, BSc
Maria-Kuhn-Gasse 6/3/18, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

Mit dem Ende dieser Diplomarbeit schließe ich mein Studium ab und es ist an der Zeit all jenen die bei eben diesem mitgewirkt haben zu danken. Bevor ich aber dazu komme, möchte ich mich herzlichst bei meinem Betreuer Herrn Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger für die bestmögliche Betreuung die sich ein Student nur wünschen kann bedanken.

Großer Dank gilt meinen Eltern für die fortwährende Unterstützung und die Finanzierung meiner Bildungslaufbahn. Obendrein möchte ich mich auch bei Wolfgang Wallner, BSc für die unzähligen gemeinsam gemeisterten Herausforderungen bedanken. Nicht unerwähnt bleiben soll auch Martina Lang, MA, die mich viele Jahre während meines Studiums motiviert hat und somit auch zu diesem Abschluss ihren Beitrag geleistet hat. Überdies gilt mein Dank all meinen Freunden, die neben Vollzeitarbeit und Diplomarbeit für die notwendige Abwechslung gesorgt haben.

Abschließend noch ein herzliches Dankeschön an Herrn Dr. Bernd Müller von der Robert Bosch GmbH für den Auftrag zu der nun vorliegenden Arbeit.

Abstract

A promising approach for lowering the complexity of safety critical embedded systems like, e.g., the distributed network of Electronic Control Units (ECUs) in a modern car, is to combine the functionality of multiple ECUs in one unit. Doing so, however, gives rise to the problem of mutual influence of otherwise independent functionalities. Hence, proper separation in between them becomes critical and mandatory. One foundation for this is to introduce a hardware enforced memory separation.

This work is going to describe two different memory protection mechanisms, i.e., the hardware enforced separation in multiple memory regions by a Memory Protection Unit (MPU) and the full memory virtualization by virtual memory. The latter is nowadays common in PCs and even mobile phones and is implemented in a hardware assisted fashion by using a Memory Management Unit (MMU). This work is going to put its focus on this hardware unit and its failure possibilities. To the best of our knowledge, no such in-depth elaboration about MMU failure modes exists.

The MMU features provided by a selection of modern embedded system processors are presented and described. These features are then abstracted in a theoretical MMU, termed hypothetical MMU, consisting of multiple functional blocks. The functionalities and interconnections of these blocks are explained and an in-depth analysis of the failure modes of this MMU is provided. The analysis is based on a single-fault assumption w.r.t. faults within the MMU and additionally considers wrong memory accesses by the processor in combination with MMU internal faults.

In order to confirm, extend or refute the theoretical analysis, a simulation based fault injection in the MMU of the LEON3 processor is performed. This fault injection covers all internal signals of a synthesized version of the MMU and is implemented with tool support. The results of the fault injection are analyzed for their validity and possible bias due to the implemented fault injection method.

While the quantitative forecasting of failure rates is not an objective of this thesis, it is shown that MMUs are prone to critical failure modes and thus should be appropriately protected if a usage in a safety critical system is targeted. The fault injection shows that the theoretical analysis sufficiently covers the experienced failure behavior. Furthermore, especially critical components are pinpointed with relative failure rates and different fault mitigation mechanisms are presented along with an evaluation of their feasibility.

Kurzfassung

Ein erfolgversprechender Ansatz, um die Komplexität sicherheitskritischer Systeme – wie zum Beispiel dem verteilten Netzwerk von Steuergeräten in einem modernen Auto – zu reduzieren, ist es die Funktionalität mehrerer Steuergeräte auf eines zusammenzufassen. Dadurch ergibt sich allerdings die Möglichkeit, dass sich andernfalls unabhängige Funktionalitäten gegenseitig beeinflussen. Deshalb wird in diesem Fall eine zuverlässige Abgrenzung zwischen den Funktionalitäten unverzichtbar. Eine Grundlage hierfür ist eine in Hardware realisierte Speicherabgrenzung.

Diese Diplomarbeit beschreibt zwei unterschiedliche Speicherabgrenzungsmöglichkeiten. Dies sind einerseits die Hardware-realisierte Unterteilung in verschiedene Speicherregionen mittels einer Memory Protection Unit (MPU) und andererseits die vollständige Speichervirtualisierung durch virtuelle Speicherverwaltung. Letzteres ist der heutzutage übliche Ansatz in PCs sowie Smartphones und wird durch eine sogenannte Memory Management Unit (MMU) Hardware-unterstützt implementiert. Diese Arbeit wird ihr Hauptaugenmerk auf diese Hardwareeinheit und ihre Fehlermöglichkeiten legen. Uns ist bisher keine derartige Ausarbeitung über MMU Fehlermöglichkeiten bekannt.

Die MMU Funktionen einer Auswahl moderner Prozessoren für eingebettete Systeme werden dargestellt und beschrieben. Auf Basis dieser Funktionen wird eine theoretische MMU, bestehend aus unterschiedlichen Blöcken, abstrahiert. Die Funktionen und Querverbindungen dieser Blöcke werden erklärt und auf ihre Fehlermöglichkeiten untersucht. Die Analyse beruht auf einer Einzelfehlerannahme bezüglich Fehler innerhalb der MMU und behandelt zusätzlich noch Kombinationen aus illegalen Speicherzugriffe des Prozessors mit internen Fehlern der MMU.

Um diese theoretische Analyse zu bestätigen, zu erweitern oder zu widerlegen, wird eine simulationsbasierte Fehlereinstreuung in die MMU des LEON3 Prozessors durchgeführt. Diese Fehlereinstreuung deckt alle internen Signale einer synthetisierten Version dieser MMU ab und wird automatisiert durchgeführt sowie ausgewertet. Die Ergebnisse der Fehlereinstreuung werden auf ihre Validität und mögliche systematische Messabweichungen untersucht. Des Weiteren werden die Ergebnisse der theoretischen Analyse zugeordnet.

Während die quantitative Vorhersage der Fehlerraten kein Ziel dieser Arbeit ist, wird gezeigt dass MMUs anfällig für kritische Fehlermöglichkeiten sind und deshalb ausreichend Schutzmaßnahmen getroffen werden müssen, wenn ihr Einsatz in sicherheitskritischen Systemen angestrebt wird. Überdies zeigt sich, dass die theoretische Analyse das festgestellte Fehlerverhalten ausreichend abdeckt. Zusätzlich werden besonders kritische Teile mit relativen Fehlerraten offenbart und verschiedene Schutzmaßnahmen beschrieben, sowie auf ihre Anwendbarkeit überprüft.

Contents

Contents ix

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Aims and Scope . . . 2
1.3 Terminology and Background . . . 2
1.3.1 Faults, Errors and Failures . . . 2
1.3.2 Transient Faults . . . 3
1.4 Methodical Approach and Structure . . . 4

2 Overview of Memory Management and Protection Approaches **5**
2.1 Memory Protection Units . . . 5
2.2 Virtual Memory . . . 6
2.2.1 Paging . . . 6
2.2.2 Memory Protection . . . 9
2.2.3 Memory Sharing . . . 9
2.2.4 Performance Optimizations . . . 10
2.2.5 Multilevel Page Tables . . . 11
2.2.6 Timing Uncertainty of Virtual Memory . . . 13
2.3 Function Blocks . . . 14

3 Synopsis of Existing Memory Management Units Implementations **17**
3.1 Memory Management Units of the ARM Architecture represented by ARM Cortex A9 and Cortex A15 . . . 20
3.1.1 Cortex A9 . . . 20
3.1.2 Cortex A15 . . . 22
3.2 Memory Management Units of the Analog Devices Inc. Blackfin Architecture represented by Analog Devices Inc. ADSP-BF54x . . . 24
3.3 Memory Management Units of the Power Architecture represented by Freescale Semiconductor e200z4 . . . 25
3.4 Memory Management Units of the SPARC Architecture represented by Aeroflex Gaisler LEON3 and LEON4 . . . 26
3.5 Memory Management Unit of the Altera Corporation NIOS II Architecture . . . 27

4 Failure Mode Analysis of a Hypothetical Memory Management Unit 31
4.1 Fault Hypothesis . . . 31
4.2 Description of Function Blocks . . . 32
4.2.1 Considered MMU Functions . . . 33
4.2.2 Virtual Page Number Extraction (VPNE) . . . 33
4.2.3 Optional: Translation Lookaside Buffer (TLB) Replacement Strategy (TRS) . . . 34
4.2.4 TLB Storage Element (TSE) . . . 35
4.2.5 Hit Selector (HS) . . . 37
4.2.6 Optional: Hardware Table Walk (HTW) . . . 38
4.2.7 TLB Manager (TM) . . . 39
4.2.8 Fault Handler (FH) . . . 42
4.2.9 Address Merger (AM) . . . 42
4.2.10 Optional: Memory Interface (MI) . . . 43
4.3 Failure Effects Involving Multiple Function Blocks . . . 44
4.3.1 Fault Vector from HTW to FH and TM . . . 44
4.3.2 TLB Entry from HS to AM, FH and TM . . . 45
4.3.3 Hit indicator from HS to TM and FH . . . 45
4.3.4 Virtual Page Number from TM to TSE and HTW . . . 46
4.3.5 Various Control Data from TM to TSE and HTW . . . 46
4.3.6 Supervisor Access Indicator from TM to TSE and FH . . . 46
4.4 Potential Failure Propagation . . . 46
4.4.1 Output Failures after AM Block . . . 46
4.4.2 Output Failures after FH Block . . . 48
4.4.3 Output Failures after MI block . . . 50
4.4.4 Intermediate Failure Effects . . . 50
4.5 Conclusion of the Failure Mode Analysis . . . 59

5 Fault Injection Experiments 61
5.1 Goals of the Fault Injection . . . 61
5.2 Method . . . 62
5.3 Considerations and Design Decisions . . . 62
5.4 Tools . . . 64
5.4.1 LEON3 MMU Test Bench . . . 65
5.4.2 Fault Injection GUI . . . 68
5.4.3 Result Comparison GUI . . . 68
5.5 Parameters . . . 75
5.5.1 Workload of the Test Bench . . . 75
5.5.2 Fault Injection Sequence . . . 79
5.6 Result Evaluation . . . 82
5.6.1 Influence of the Workloads . . . 82
5.6.2 Influence of the Workload Blocks . . . 84
5.6.3 Detected First Faults per Indicated Operation . . . 87

5.6.4 Special Cases due to the Fault Injection . . . 89
5.6.5 Consequences of Deviating Actions . . . 90
5.6.6 Consequences of Deviating Data . . . 96
5.7 Results and Conclusion . . . 97

6 Safety Enhancement Possibilities for Memory Management Unit 101
6.1 Possibilities for Error Mitigation . . . 101
6.1.1 Parity . . . 101
6.1.2 Dual Rail Encoding . . . 102
6.1.3 Single-Error Correction and Double-Error Detection (SEC-DED) . . . 102
6.1.4 SRAM Protection through Built-In Current Sensors (BICSs) . . . 102
6.1.5 Hardened Latches/Flip-flops . . . 103
6.1.6 Hardware Redundancy . . . 103
6.1.7 State Machine Protection . . . 103
6.1.8 Time Redundancy Implemented in Software . . . 103
6.2 Relevant Mechanisms for the Implementation of a Safety Enhanced MMU . . . 103
6.3 General approach for Protection of a MMU . . . 104
6.4 Analysis of Protection Approaches for the LEON3 MMU . . . 105
6.4.1 Protection Approach with Duplication of the complete MMU besides the Content Addressable Memory (CAM) Modules . . . 105
6.4.2 Duplication of the Complete MMU . . . 105
6.4.3 Protection of the Random-Access Memory (RAM) Portion and Duplication of Everything else . . . 106
6.4.4 Other Optimization Possibilities . . . 106
6.5 Conclusion . . . 106

7 Results and Conclusion 107

A E-Mail Correspondence 109

Acronyms 111

Bibliography 113

CHAPTER 1

Introduction

1.1 Motivation

Modern cars use a high count of ECUs in order to clearly separate different, possibly safety critical, functions from each other. For example, new cars in the year 2009 used 20 to 70 such ECUs [EJ09]. Todays microprocessors are, from a performance point of view, easily capable of handling the processing effort of multiple functions in parallel. Nevertheless, without a proper boundary between those functions, multiple functions in one ECU introduce a possibility of mutual influence. This is especially problematic if functions of different criticality levels are involved. In this case, the functionalities with lower criticality would have to be developed according to the standards applicable to the ones with the highest criticality, because they may influence these functionalities. On the other hand, reduction of ECU count in a car would allow to cut down hardware costs, power consumption and weight.

A possibility to overcome the problem of mutual influence, is to provide a virtual environment for each separate function. Thus, the separation needs to be considered as critical as the functions with the highest criticality in the system. However, the implementation of the virtual environment may be realized as part of an Operating System (OS) and thus can be reused. This allows to ease the processes of certification, function development as well as to ensure error containment, with reasonable cost efficiency. In order to ease the function development, the virtual environment should manage the resource sharing in a way, that it is transparent for the functions.

Memory management is one central concept for that matter. It targets a clear memory separation between multiple functions. First of all, this allows easier detection of programming errors, because whenever one of the functions tries to access a memory region it is not allowed to, a memory exception may be raised. Secondly, this kind of separation is a must in order to allow multiple functions with different criticality levels on the same hardware resource. One possibility to ensure a clear memory separation is to use virtual memory management.

In order to allow virtual memory management with reasonable performance, typically a hardware unit, termed Memory Management Unit (MMU) is used. Virtual memory is a common technique for hardware enforced process separation in modern PCs. With the rise of processing

power, as well as decreasing cost per transistor, this technique nowadays has even made its way on mobile phones. Nevertheless, it is still not common in safety critical environments.

1.2 Aims and Scope

This thesis targets the advantages and drawbacks of using a MMU in a safety critical context. Therefore, the focus of the drawbacks is on the potential risks that are introduced by using this hardware mechanism.

Performance enhancing mechanisms for MMUs are mentioned but not evaluated in detail. Statistical failure rates of MMU implementations in commercial microprocessors are not provided in this thesis. Instead, an in depth theoretical analysis of failure possibilities by MMUs is provided. The theoretical analysis is backed up by simulated fault injection experiments in a Register-Transfer Level (RTL) model of a commercial MMU. Furthermore, possible countermeasures against the risks are mentioned and theoretically evaluated. To the best of our knowledge no such in depth evaluation of MMUs w.r.t. to safety critical systems exists.

1.3 Terminology and Background

This section is going to establish the used terminology for faults, errors and failures. Furthermore, the different fault types are introduced and the main causes for transient faults are described.

1.3.1 Faults, Errors and Failures

We use the terms fault, error and failure with the meaning presented in [ZAV04]:

- *Fault is a physical defect, imperfection, or flaw that occurs within some hardware or software component.*
- *Error is a deviation from accuracy or correctness and is the manifestation of a fault.*
- *Failure is the non-performance of some action that is due or expected.*

Furthermore, the types of faults are commonly categorized in the following classes presented in [ZAV04]:

- *Permanent faults: Caused by irreversible component damage, such as a semiconductor junction that has shorted out because of thermal aging, improper manufacture, or misuse.*
- *Transient faults: Triggered by environmental conditions such as power-line fluctuation, electromagnetic interference, or radiation. These faults rarely do any lasting damage to the component affected, although they can induce an erroneous state in the system. According to several studies, transient faults occur far more often than permanent ones, and are also far harder to detect.*

- *Intermittent faults: Caused by unstable hardware or varying hardware states. They can be repaired by replacement or redesign.*

Permanent faults are detectable by appropriate test vectors and their effects can thus be minimized by, e.g., frequent self tests. According to [Con02] errors induced by intermittent faults are very similar to those caused by transient faults. The main differences are that they occur repeatedly at the same location and can be removed by replacement of the defective circuit.

Thus, we can conclude that the transient fault type is the most difficult type to detect and also may lead to the same effects experienced by intermittent faults. We are going to focus on this fault type within this work.

1.3.2 Transient Faults

Transients faults are caused by the environment of the system under consideration. For some causes like, e.g., power jitter or electromagnetic interference, well known countermeasures exist and thus the failure rate can be optimized until the targeted reliability is reached.

On the other hand, transient faults caused by Single Event Effects (SEEs) cannot be handled without major drawbacks. Shielding, e.g., is not an option to decrease the Soft Error Rate (SER) of portable devices as stated by [Bau05]: *while the SER due to cosmic neutrons of a system operating in a basement surrounded by many feet of concrete could be significantly reduced [...] for personal desktop applications or portable electronics, little can be done to reduce the cosmic ray portion of the SER*. To the best of our knowledge, every mitigation mechanism either introduces a performance penalty or introduces a substantial area overhead w.r.t. chip size. Some of these mitigation possibilities are introduced in chapter 6.

Transient faults due to SEEs, e.g., an ionized particle strike, can be categorized in the following classes:

- Single Event Transient (SET): The event causes a voltage pulse on a signal line. Note that in Complementary Metal Oxide Semiconductor (CMOS) technology the voltage pulse can be of both digital values, depending on the hit transistor type. Typical pulse widths stated in literature vary from tens of picoseconds [LKJ+12] to two nanoseconds [GAN+10]. On the one hand, the wide range of differing data stems from the dependence of the pulse width on the technology node. On the other hand, [GAN+10] states that the variance is additionally influenced by the measurement circuits due to pulse broadening effects and structural differences of the well contact.

  SETs may be masked out by one of the following mechanisms:

  - Electrical masking: The SET is attenuated below the threshold voltage of the observing element and thus has no consequences.
  - Logical masking: The SET is masked out in combinational logic, e.g., a positive voltage spike on one of the inputs of an AND-gate has no consequences if one of the other inputs is constantly inactive.

- Temporal masking: In synchronous logic, SETs in the combinational paths before a flip-flop are masked out, if they occur outside the setup and hold window. Hence, the SET pulse width to clock cycle ratio is an important factor.

- Single Event Upset (SEU): A SET may be latched by a storage element if it occurs during the latch window and thus causes an unintended state. Furthermore, the SET may occur on a clock line and cause a flipflop to latch its input asynchronously. Additionally, it is also possible that the state of a memory element is directly inverted by an event.
- Single Event Functional Interrupt (SEFI): According to [jes06] a SEFI is a *soft error that causes the component to reset, lock-up, or otherwise malfunction in a detectable way, but does not require power cycling of the device (off and back on) to restore operability*.
- Single Event Latch-up (SEL): In case the event triggers a CMOS parasitic bipolar transistor it leads to a latch-up, i.e., an abnormal high-current state. In order to remove the latch-up power cycling of the device is necessary and it is also possible that the device is irreversibly damaged due to the power dissipation caused by the high current. Thus, if no countermeasures are taken, a SEL may lead to a permanent failure.

1.4 Methodical Approach and Structure

This thesis starts with a general overview of memory management and protection approaches in chapter 2. After this overview, it focuses on the MMUs itself. First the MMU features of a variety of processors for embedded systems are listed in chapter 3. Secondly a theoretical MMU is analyzed for its failure mode possibilities in chapter 4. Chapter 5 verifies the theoretical analysis by simulated fault injection experiments in the MMU of the LEON3 processor. Furthermore, chapter 6 gives a overview and a quick evaluation of various protection mechanisms. Finally, the results of this thesis are summarized in chapter 7.

CHAPTER 2

Overview of Memory Management and Protection Approaches

Whenever a single processor has to run multiple processes, using the same physical memory, the possibility of mutual influence of those processes arises. If no countermeasures are taken, one process could manipulate the part of memory where another process resides, thus leading to malfunction of the other process. This type of error can happen intentionally by a malicious attacker or accidentally by a programming error or a soft error. The consequences, especially in safety-critical environments, tend to be severe. Without any protection mechanism a simple process may crash the OS and thus can cause a complete system failure. Even worse, it can cause false outputs to possibly attached actuators or other systems.

To overcome this problem, the memory access of the individual processes has to be restricted. This can either be achieved by assigning access permissions to memory regions and enforcing those by a MPU, or by assigning each process its own virtual address space and enforcing of access permissions at the translation to physical memory addresses, a technique known as virtual memory. This chapter gives an overview of those two techniques.

2.1 Memory Protection Units

Using physical memory addresses has some major drawbacks, especially if more than one process resides in memory. If an active process can access every physical memory location, it can easily thrash any other process in memory – either intentionally or by accident. Furthermore, it has to be ensured that every process uses correct addresses, independent from where it is loaded in memory. This relocation can for example be achieved by the linker, if all processes in memory are known in advance, or at execution time by a relocating loader. Another possibility is to tell the compiler to always use relative addresses which leads to position-independent code [Tan09, p. 223].

A MPU allows to separate the memory into regions, where for each region memory access permissions can be set. Those permissions are then enforced by the MPU-hardware. Nevertheless, processes have to use physical addresses and therefore some kind of relocation is still needed.

If a memory access fails because of insufficient permissions, it is not passed on to the memory and an exception is raised. The exception can then be handled by the OS. Memory regions are specified by a start address and a region-size. Typical implementations of MPUs allow only a small number of memory regions to be configured, e.g., the MPU of Altera Corporation NIOS II allows up to 32 instruction regions and 32 data regions [Alt11, p. 2-16].

In general those memory regions can overlap and the applied permissions are determined by some kind of priority scheme. E.g., the MPU of the ARMv7-R architecture from ARM uses a priority value for each region and if two regions overlap, the permission bits of the region with the highest priority are used. This allows to lower the number of separate regions needed. It is possible to use one region that covers the whole memory for the OS and separate regions for different processes. The memory between and outside those process regions is thus covered by the OS-region, without specifying additional regions for every such memory segment.

The main advantage of MPUs is that they provide fully deterministic timing behavior. Furthermore, the complexity of their hardware implementation as well as the corresponding software part, to manage the protection, is lower [ARM11].

2.2 Virtual Memory

The following discussion relies on [Tan09], [HP09] and [HP12]. Please refer to the original books for more in-depth information.

Virtual memory is a technique that provides every process with its own contiguous virtual address space consisting of so-called virtual addresses and thus gets rid of the need for relocation. Those virtual addresses correspond to locations in physical memory that are addressed by so-called physical addresses. The virtual addresses of different processes can be the same and there is no restriction on the ratio between the virtual and the physical address space. Furthermore, virtual memory allows to specify fine grained access permissions for the virtual address space on the basis of small memory regions, e.g., with 4 KB size. Additionally, it is possible to use more virtual memory than physical main memory is available.

The mapping from virtual addresses to unique[1] physical addresses is managed by the OS. The actual address translation is typically supported by hardware, the so-called MMU, although pure software approaches like, e.g., [CG05] for MMU-less processors also exist.

2.2.1 Paging

The virtual address space is partitioned into fixed size chunks, called pages, that correspond to page frames of equal size in physical memory. The mapping of a virtual page address to a page frame address is stored in the page table, which in turn is managed by the OS. Every process has its own page table. Only those virtual pages that are mapped take up space in physical memory,

[1] except for sharing memory contents among multiple processes

beyond the size the entry requires in the page table anyway. Virtual addresses consist of an address part, that indexes the corresponding page table entry, and an offset part, that indexes a specific memory location within the page. The page table is also loaded in main memory. An example address translation from 16-bit virtual addresses to 15-bit physical addresses is given in figure 2.1.

To speed up the address translations by the MMU a fast cache for address translations, the TLB, is used. Whenever an accessed entry does not reside in the TLB, this case is termed TLB-miss, it has to be fetched from the page table. This is called page table walk. In case that the MMU provides a hardware implemented page table walk, the page table entries of the OS have to be formated in compliance with the format understood by the MMU. An exception to this rule is, if the MMU uses a software managed TLB and thus relies on a software implemented page table walk by the OS. In that case the entries can be reformatted when they are written to the TLB and thus may contain additional information that is only used by the OS. As the TLB is a hardware cache, the format is fixed from that point on.

By using a 12-bit offset, as in the example of figure 2.1, it is possible to address page sizes of 4 KB on byte granularity. The actual page size is in principle determined by the capabilities of the MMU, although the OS may combine multiple pages to a bigger one as mentioned in [Tan09, p. 217]. Some MMUs provide the possibility to select between multiple possible page sizes, e.g., ARM Cortex A9 allows to select between 4 KB, 64 KB, 1 MB, 16 MB and Freescale Semiconductor e200z4 even provides 23 different sizes. This allows to tweak the address translation for a specific application, as the page size is a trade-off between the number of TLB-misses and memory usage. As stated in [HP12, p. B-47], a larger page size means fewer TLB misses and for some programs, TLB misses can be as significant on Cycles per Instruction (CPI) as cache misses. A smaller page size means better memory usage, because pages are always reserved completely. Nevertheless, the page table size grows inversely proportional to the page size.

It is not necessary to hold all page frames of a given process in main memory. Page frames can be moved to a cheaper, bigger but slower secondary storage[2] to free main memory. Nowadays this is also referred to as swapping, although swapping in the traditional sense meant that the whole process is swapped out to secondary storage. Which pages are getting moved depends on the page replacement strategy used by the OS.

Another memory-saving possibility is that whenever a program is to be fetched from persistent memory only the first page is loaded in main memory. This is called demand paging.

Pages that do not reside in main memory are marked as invalid in the page table. Whenever such a page is accessed a page fault occurs. It is the operating system's responsibility to load the corresponding page frame from secondary storage to main memory and restart the faulting operation. Note that, due to the lower speed of secondary storage, this is a fairly slow operation. However, most processes exhibit a locality of reference, which means that they reference only a small part of their pages during any phase of execution [Tan09, p. 207]. Once all pages that are used during an execution phase are in main memory, those processes run without page faults.

As already mentioned, the ratio between the virtual and the physical address space is not restricted. Hence, it is possible to use, e.g., virtual addresses of 32-bit and physical addresses of 36-bit, as it is done by the Physical Address Extension (PAE) of 32-bit x86 processors. This

[2] sometimes also called backing storage

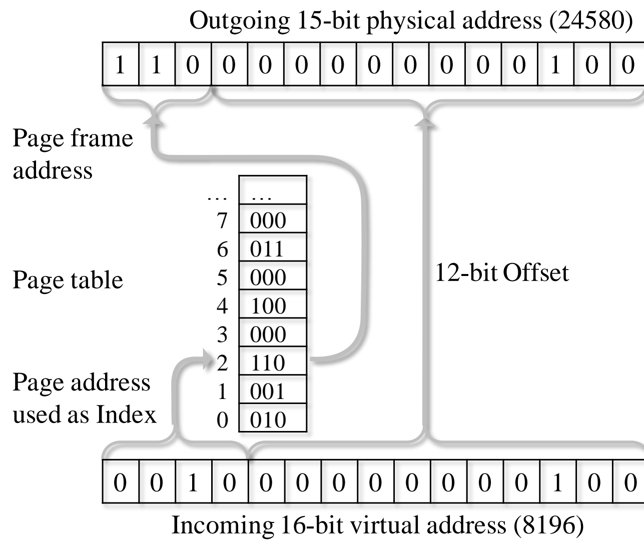

Figure 2.1: Example address translation from 16-bit virtual addresses to 15-bit physical addresses

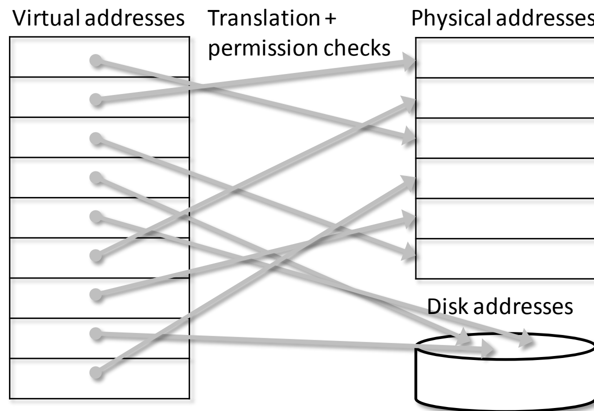

Figure 2.2: Mapping from virtual addresses to physical addresses, where some pages are swapped out on secondary storage

allows a single process to use up to 4 GB of main memory (32-bit virtual addresses), although it is possible to address 64 GB of physical memory. The whole main memory can thus be only fully exploited by multiple processes.

It is also possible to use a virtual address space that is bigger than the physical one. Consider, e.g., a 32-bit x86 processor with less than 4 GB RAM. If in that case more virtual pages are used than there is space in main memory, some pages are getting swapped out on secondary storage, as depicted in figure 2.2. Which pages are getting swapped out depends on the page replacement

Permission	Description
Read	Allow read accesses from this page
Write	Allow write accesses to this page
Execute	Allow execution of code at this page
Cacheable	Allow caching of this page

Table 2.1: Typical permission bits for memory protection

strategy used by the OS.

There can also be multiple processes in main memory, where the sum of the pages exceeds the number of physical pages possible in main memory. In that case some pages are getting swapped out on secondary storage as well.

2.2.2 Memory Protection

In order to restrict memory access in certain ways, the MMU provides mechanisms to specify access permissions at page level. Those access permissions are then enforced at every address translation, before the memory access is passed on to the physical memory. In case of an access violation no memory access is performed and the OS is invoked.

The access permissions are specified by access bits for every page table entry. The capabilities of the access restrictions vary between different MMU implementations. In general there are possibilities to restrict read and write access. Some MMUs allow to mark pages as not executable to prevent the processor from executing instructions from memory regions that are only used for data storage. This is a potential countermeasure against buffer overflow attacks. The typical permission bits are summarized in table 2.1. MMUs also allow to specify at least two different access permissions for every page, depending on whether the page access is being attempted by user or supervisor software (e.g., a system call).

2.2.3 Memory Sharing

In order to save memory and optimize performance, multiple processes may share some pages that contain data or code. This can be achieved by having virtual addresses in the page tables of the processes pointing to the same physical page frame, which is clearly more efficient than having the same page loaded multiple times but comes with some pitfalls.

Pages that are read only to all processes, e.g., code pages, can easily be shared. Nevertheless if one of the processes terminates, it has to be ensured that the shared pages – that are still in use by the other processes – stay in memory. Pages that are also writable by some processes can either be used for fast data exchange between the processes, e.g., shared memory, or have to be duplicated. The latter case can be optimized by a technique called copy on write. For example the fork system call, as provided by UNIX based systems, can be implemented by creating a new process with its own page table but with the same set of pages and corresponding page frames. Hence, there is no copying of pages needed at the time the fork call is made. However, all the pages need to be marked as read only for both processes. If one process tries to write to a page,

the protection alerts the OS. The OS then creates a copy of the page, replaces the page table entry of one of the processes and allows writes to both of the pages. As now every process has its own copy of the page, writes can be safely performed without influencing the other process. Therefore, only pages that are actually written need to be duplicated. [Tan09, p. 220 f]

2.2.4 Performance Optimizations

Due to the fact that it would be slow and inefficient to have the operating system performing every address translation in software, this task is performed by specialized hardware – the so called MMU. Essentially every memory reference of the processor is passed through the MMU, where it is translated to a physical memory reference. If this translation would be done in software, it would be necessary to access the memory at least twice on each memory access, because the translation involves a lookup of the physical address in the page table which in turn resides in memory. This would imply a huge performance penalty.

To overcome this bottleneck a specialized hardware, the MMU, is used in conjunction with a fast translation buffer, the TLB. The TLB can be seen as a cache for address translations and its content is either managed by software or hardware. Every field of an entry in the TLB has a one-to-one correspondence with the fields of the related entry in the page table, except for the virtual address. The necessity of this field depends on the associativity of the TLB (see section 2.2.4.1). Every time a TLB miss occurs, the entry has to be fetched by a page table walk and the failed instruction needs to be restarted afterwards. According to [HP09, p. 503] typical values for a TLB are:

- Size: 16 – 512 Entries
- Hit time: 0.5 – 1 clock cycle
- Miss penalty: 10 – 100 clock cycles
- Miss rate: 0.01% – 1%

As every process has its own virtual address space and thus page table, the TLB contains entries of the previously active process after a context switch. In those the virtual addresses may map to different physical addresses. Hence, in order to use correct page table entries after a context switch, some MMU implementations rely on the OS to flush the TLB at every context switch, which is fairly time-consuming but lowers the complexity of the hardware. Others also store the Process Identifier (PID) of the page in the TLB and use the PID during address translations. Those implementations thus do not depend on a complete flush. This is also advantageous if the process is reactivated and some of its page table entries are still stored in the TLB. All processors that are dealt with in this thesis and that support virtual addresses use such a PID field. Additionally there may be a possibility to mark an entry as global, what means that it is available for all processes.

2.2.4.1 TLB Associativity

The page address is used to find the corresponding page table entry in the TLB. There are three possibilities to determine the actual entry:

- Direct mapped: The position within the TLB can be calculated by:

  $$\text{position} = (\text{page address}) \text{ modulo } (\text{number of TLB entries})$$

  The advantage of direct mapped is that no search of entries is needed, however this comes with the price of a higher miss rate.

- Fully associative: The entry can be placed at an arbitrary position within the TLB. Thus, all the entries have to be searched for the correct one. The characteristics of fully associative are the opposite of direct mapped. The hardware complexity is higher due to the search but the miss rate is lower.

- n-way set associative: The set within the TLB can be calculated by:

  $$\text{set} = (\text{page address}) \text{ modulo } (\frac{\text{number of TLB entries}}{\text{number of sets n}})$$

  Within the set the correct entry has to be searched. This technique can be used to implement a compromise between the two extremes direct mapped and fully associative.

In case that the divisor is a power of 2, the modulo operation can be performed by using the lowest $\log_2(\text{divisor})$ bits of the page address.

According to [HP09, p. 479] fully associative caches are searched in parallel by using a comparator for each cache entry and thus making fully associative placement only reasonable for caches with a small number of entries. As a TLB is a cache, this also applies here. However, TLBs typically only contain a small number of entries and therefore often use fully associative placement, e.g., Aeroflex Gaisler LEON3 uses a fully associative TLB-structure [Aer12, p. 726], whereas ARM Cortex A15 uses fully associative level 1 TLBs and a 4-way set associative level 2 TLB [ARM12a, p. 1-5].

2.2.5 Multilevel Page Tables

In its simplest form a page table has to contain an entry for every page of the virtual address space, either with a valid physical address or marked as an invalid entry. Consider a 32-bit virtual address space and a quite common page size of 4 KB. This results in a page table with 2^{20} entries, leading to a memory usage of 4 MB, if every entry takes 4 byte. In case of a 64-bit virtual address space the overhead gets much bigger, because every page table would occupy 16 PB, if we still consider 4 byte entries. However, in most cases only a minor fraction of the entries is actually used. As every process has its own page table, this implies a huge waste of memory.

The problem can be solved by using multilevel page tables. The virtual address is split up in parts, such that each part indexes a page table of the corresponding level. Consider again a

10 Bit	10 Bit	12 Bit
Top-Level	Second-Level	Offset

Figure 2.3: Example of an address splitting at multilevel page tables

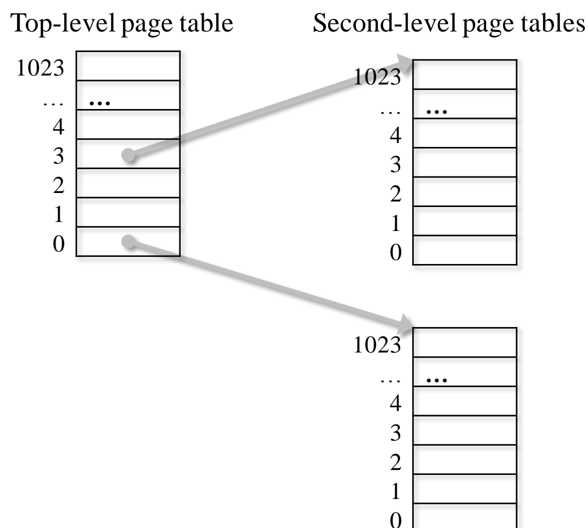

Figure 2.4: Two-level page tables

32-bit virtual address space and a page size of 4 KB. The resulting 20-bit page address, e.g., can be split up in two 10-bit parts (see figure 2.3). At every address translation the MMU first performs a lookup in the top-level page table, using the most significant bits. The top-level page table contains an entry for the second-level page table (see figure 2.4). Thus, the MMU uses the result of the first lookup to perform a lookup in the second-level page table and finally outputs the page frame address. In this example the top-level page table and every second-level page table contains 1024 Entries. The advantage lies in the fact that only those second-level page tables need to be present in memory, that are actually used.

Suppose a process needs as a whole 12 MB of memory, 4 MB for program text, 4 MB for data and 4 MB for the stack. Each of the three segments are contiguous but between them are many unused virtual addresses. Using the above two-level page table, it suffices to hold four pages, each with 1024 entries, in memory [Tan09, p. 198]. Thus, reducing the memory consumption of the page tables from, e.g., 4 MB to 16 KB.

Multilevel page tables are widely used. E.g., x86 Processors use two levels, x86 with PAE use three levels and x86 with the 64-bit extension AMD64 or Intel 64 use four levels [AMD12, p. 124 ff].

Note that multilevel page tables can also be used to implement multiple page sizes. The top-level page table with 1024 entries effectively separates the 4 GB virtual address space in 4 MB

chunks. Thus, if the top-level page table entry is directly used as part of the physical address a single page table lookup suffices and the corresponding page size is 4 MB. To allow two page sizes the top-level entries have to be marked as either page table entries (4 KB page size) or page frame entries (4 MB page size). The basic principle is depicted in figure 2.5 and, e.g., used at ARM Cortex A9 and Cortex A15 [ARM11, p. 2555 ff].

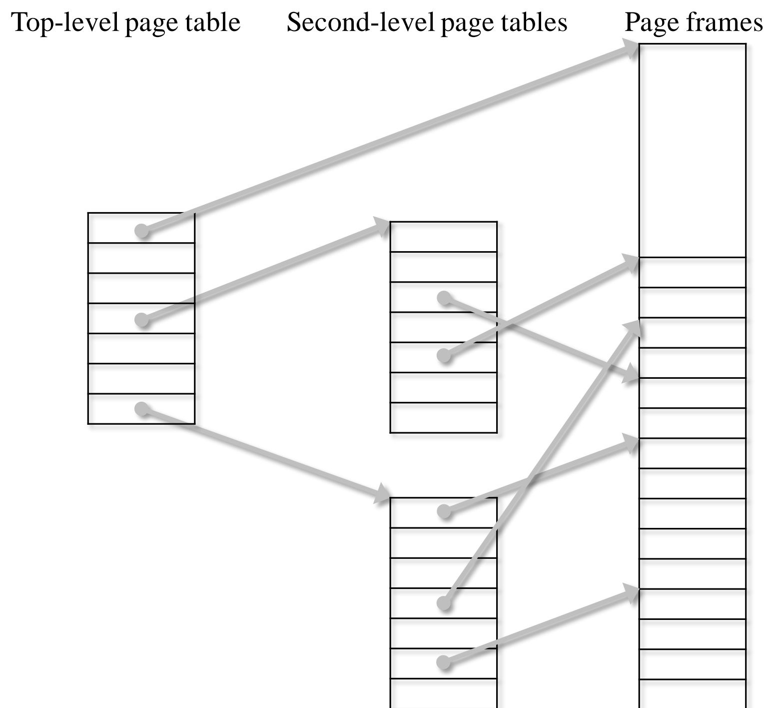

Figure 2.5: Multiple page sizes by using multilevel page tables

2.2.6 Timing Uncertainty of Virtual Memory

Virtual memory has some implications with respect to real-time systems because it is optimized for average performance instead of worst case performance. The first uncertainty arises by accessing the TLB, an entry may not reside in the TLB which leads to an extra memory access to fetch the entry from the page table. Sometimes the TLB is even split up in multiple TLBs and organized in a hierarchy, as it is the case at, e.g., ARM Cortex A9 and Cortex A15 processors. At this processors the TLB is split up in a first level instruction TLB, a first level data TLB and a second level unified TLB. The second uncertainty arises by accessing the page, because the page may not reside in main memory and therefore has to be fetched from secondary storage. Depending on the speed of the secondary storage, this may introduce a huge delay.

2.2.6.1 Possible Countermeasures

Some processors, e.g., ARM Cortex A9, allow to specify that a particular entry should be kept in the TLB. Nevertheless, the number of such entries is limited by the size of the TLB and sometimes even further to a subset of the TLB entries. The aforementioned processor only allows to lock four entries in the unified TLB and thus the (smaller) uncertainty of a miss in the instruction or data TLB remains. That is because the unified TLB only catches misses in the other TLBs.

In case of a software managed TLB some entries can be locked by the OS. Furthermore, the OS can also reduce the number of page faults or even eliminate them at all. However, the real-time capabilities of virtual memory are out of focus of this thesis and will therefore not be further discussed.

2.3 Function Blocks

Figure 2.6 shows the basic function blocks that have to be implemented by a MPU.

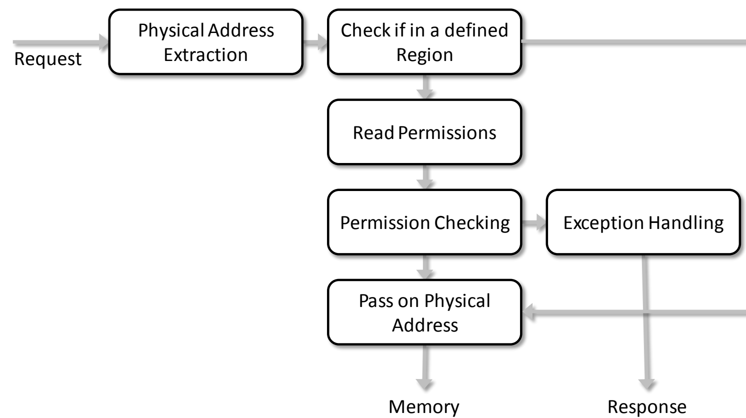

Figure 2.6: Function blocks of a MPU

Figure 2.7 shows in contrast the basic function blocks that have to be implemented by a MMU. These blocks are the starting point of the error possibility analysis in chapter 4 and are going to be discussed in more detail in that chapter.

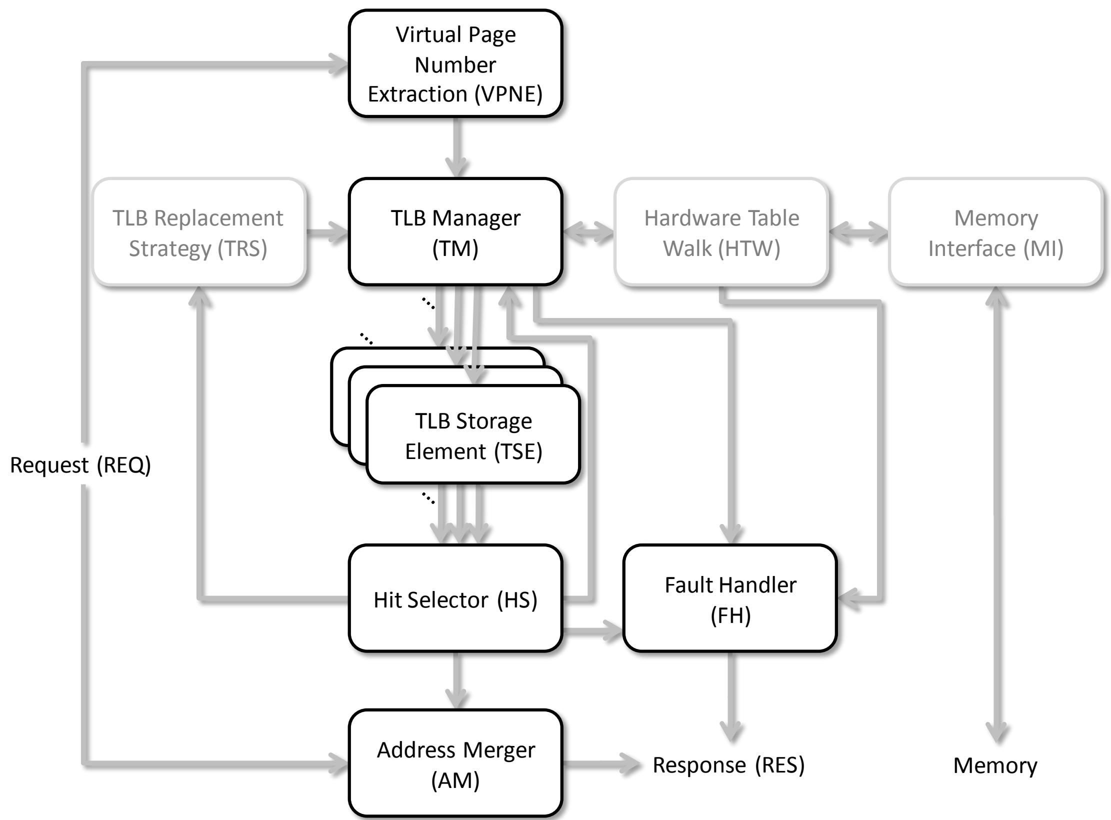

Figure 2.7: Function blocks of a MMU

CHAPTER 3

Synopsis of Existing Memory Management Units Implementations

This chapter is going to summarize the features of MMUs found in a selection of modern processors. The following processors are investigated:

- ARM Cortex A9 (ARMv7-A Architecture): A licensable microprocessor Intellectual Property (IP) core, designed for scalability to "*provide a wide range of market applications from mobile handsets through to high-performance consumer and enterprise products*" [1]. Up to four cores can be combined to meet high power demands.
- ARM Cortex A15 (ARMv7-A Architecture): According to ARM, "*The ARM Cortex^{TM}-A15 MPCore^{TM} processor is the highest-performance licensable processor the industry has ever seen*" [2] Its applications include smartphones, mobile computing, digital home entertainment, wireless infrastructure and low-power servers. Up to four cores can be combined within a single processor cluster and multiple such clusters can be hold coherent through Advanced Microprocessor Bus Architecture (AMBA) version 4 technology. Furthermore, it is the only investigated processor that provides hardware virtualization support.
- Analog Devices Inc. ADSP-BF54x (Blackfin Architecture): According to Analog Devices Inc., those processors are used for "*multi-format audio, video, voice and image processing, multimode baseband and packet processing, control processing, and real-time security*" [3] and "*combine a 32-bit RISC-like instruction set and dual 16-bit multiply ac-*

[1] `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`, retrieved August 21, 2012

[2] `http://www.arm.com/products/processors/cortex-a/cortex-a15.php`, retrieved August 21, 2012

[3] `http://www.analog.com/en/processors-dsp/blackfin/products/index.html`, retrieved August 21, 2012

cumulate (MAC) signal processing functionality" [4]. They optionally provide memory protection but no virtual address translation [Ana10]. Therefore – although termed MMU in the datasheet – the memory protection mechanism corresponds more to a MPU than to a MMU.

- Freescale Semiconductor e200z4 (Power Architecture): According to Freescale Semiconductor, "*the e200z4 processor family is a set of CPU cores that implement low-cost versions of Power Architecture technology*" [Fre09, p. 1-1]. Those cores are, e.g., used in microcontrollers like the Freescale Semiconductor Qorivva MPC564xL family for chassis and safety applications in automotive environments. This latter microcontroller family "*is designed to specifically address the requirements of the safety standards IEC61508 (SIL3) and ISO26262 (ASIL-D)*" [5] and uses a dual e200z4 CPU architecture.

- Aeroflex Gaisler LEON3 (SPARC Architecture): According to Aeroflex Gaisler, "*LEON3 is a 32-bit processor core conforming to the IEEE-1754 (SPARC V8) architecture. It is designed for embedded applications, combining high performance with low complexity and low power consumption*" [Aer12, p. 694]. Its VHDL source code is freely downloadable from the Aeroflex Gaisler homepage[6]. Furthermore, there exists a derivate called LEON3FT that is enhanced with fault-tolerance against SEU errors and is functionally equivalent to the standard LEON3 processor [Aer12, p. 719].

- Aeroflex Gaisler LEON4 (SPARC Architecture): Aeroflex Gaisler describes the LEON4 core as an evolution from the LEON3 with improved performance. Therefore, as it will be seen later, the features of its MMU are similar to those of the MMU of LEON3.

- Altera Corporation NIOS II (NIOS II Architecture): According to Altera Corporation, "*Altera's Nios® II processor, the world's most versatile processor, according to Gartner Research, is the most widely used soft processor in the FPGA industry.*" [7] Furthermore, there is also a safety critical version (NIOSII_SC) available. This version was developed by Altera Corporation and Hcell Engineering to be DO-254 (Design Assurance Guidance for Airborne Electronic Hardware) certifiable and a complete certifiable package can be obtained from those vendors.

Table 3.1 gives an overview of the basic properties of the MMUs available for the aforementioned processors, those are the virtual and physical address size, the page size and how the TLB entry replacement is accomplished.

[4] `http://www.analog.com/en/processors-dsp/blackfin/products/blackfin_architecture/fca.html`, retrieved September 13, 2012

[5] `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC564xL`, retrieved August 21, 2012

[6] `http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=156&Itemid=104`, retrieved August 21, 2012

[7] `http://www.altera.com/devices/processor/nios2/ni2-index.html`, retrieved August 21, 2012

Processor	Virtual	Physical	Page Size	Page table walk
Cortex A9	32-Bit	32-Bit	4 KB, 64 KB, 1 MB, 16 MB	Hardware
Cortex A15	32-Bit	40-Bit	4 KB, 64 KB, 1 MB, 2 MB, 16 MB, 1 GB	Hardware
ADSP-BF54x	N/A	32-Bit	1 KB, 4 KB, 1 MB, 4 MB	Software
e200z4	32-Bit	32-Bit	1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB	Software
LEON3 LEON4	32-Bit	36-Bit	4 KB, 256 KB, 16 MB, 4 GB	Hardware
Nios II	32-Bit	32-Bit	4 KB	Software

Table 3.1: Overview of various MMU-Properties

3.1 Memory Management Units of the ARM Architecture represented by ARM Cortex A9 and Cortex A15

Note that ARM uses the notion *translation table descriptor* for one of the following [ARM11, p. 1315]:

- "*an invalid or fault entry*"
- "*a page table entry, that points to a next-level translation table*"
- "*a page or section entry, that defines the memory properties for the access*"
- "*a reserved format*"

However, in the following discussion only the second and third meaning are relevant.

3.1.1 Cortex A9

The key features of the MMU of ARM Cortex A9 processors are described in the following list. For a more detailed summary and further explanations see [ARM12b, p. 6-1 ff].

- The processor implements the ARMv7-A MMU as described in [ARM11, p. 1299 ff].
- Support of four different page sizes through two levels of translation tables using the short-descriptor translation table format as shown in figure 3.1:
  - Supersection: 16 MB memory region, translation properties are held in the first-level table.
  - Section: 1 MB memory region, translation properties are held in the first-level table.
  - Large page: 64 KB memory region, translation properties are held in the second-level table and the entries of the first-level tables contain translation properties and pointers to the second-level table.
  - Small page: 4 KB memory region, translation properties are held in the second-level table and the entries of the first-level tables contain translation properties and pointers to the second-level table.
- 2-level TLB structure:
  - Instruction side first level TLB: Hardware configurable 32 or 64 fully associative entries.
  - Data side first level TLB: 32 fully associative entries.
  - The first level TLBs provide a lookup of the virtual addresses in a single clock cycle.
  - Unified second level TLB implemented as a combination of:
    - a fully associative, lockable array of four elements

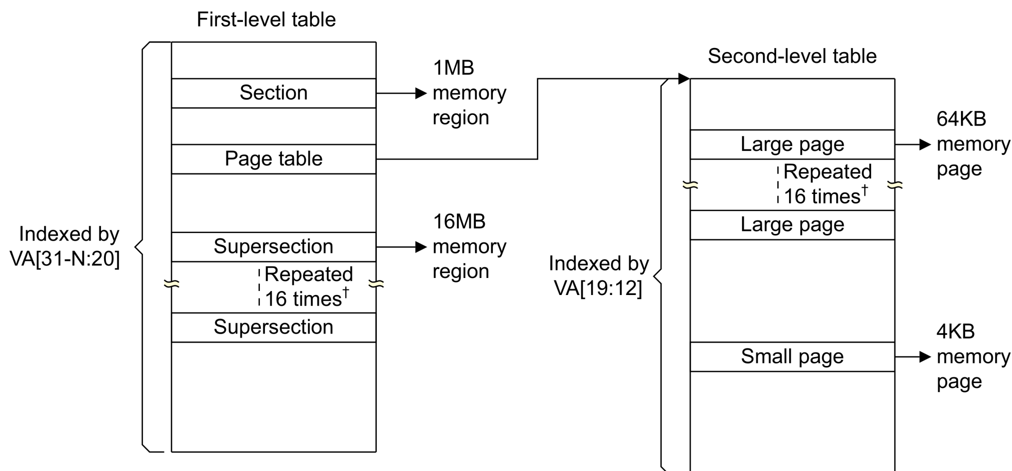

Figure 3.1: Translation table structure for address translations using short-descriptor format. Taken from [ARM11, p. 1316], slightly modified to hide unnecessary details.

  * a 2-way set associative structure of 2x32, 2x64, 2x128 or 2x256 entries

  The second level TLB catches misses from the first level TLBs. Accesses to the second level TLB take a variable number of cycles, according to competing requests from each of the first level TLBs and other implementation-dependent factors. [ARM12b, p. 6-4]

- Implements a hardware page table walk.

- Support for 16 domains, where a domain is a set of memory regions.

  Each TLB entry contains a domain field that specifies which of the domains the entry is in. First-level translation table entries for page tables and sections also include a domain field and second-level translation table entries inherit a domain setting from the first-level entry. Supersections are always domain 0 and do not include a domain field.

  For each of the 16 domains a two bit field in the Domain Access Control Register (DACR) is used to specify one of the following access permissions for every domain [ARM11, p. 1353]:

  - No access: "*Any access using the translation table descriptor generates a Domain fault.*"
  - Clients: "*On an access using the translation table descriptor, the access permission attributes are checked. Therefore, the access might generate a Permission fault.*"

  - Managers: “*On an access using the translation table descriptor, the access permission attributes are not checked. Therefore, the access cannot generate a Permission fault.* ”

- Global and address space identifiers, corresponding to PIDs.

- The following access permissions can be set in a translation table descriptor [ARM11, p. 1347]:

  - Read-only for every software.
  - Read/write for every software.
  - Read-only for supervisor software, no access by user software.
  - Read/write for supervisor software, no access by user software.
  - Additionally all descriptors for memory blocks and pages include an Execute-never (XN) bit to prohibit executing of instructions from the corresponding memory regions.
  - Furthermore, the descriptors include a field for memory region attributes to control the memory type, accesses to the cache and whether the region is shareable between processors [ARM11, p. 1357].

3.1.2 Cortex A15

For further explanations of the MMU features, besides the ones listed below, see [ARM12a, p. 5-1 ff].

- The processor implements the Extended VMSAv7 MMU as described in [ARM11, p. 1299 ff].

- Supports the Large Physical Address Extension (LPAE) for address translation of up to 40 bit physical addresses.

- Virtualization support through two translation stages with an Intermediate Physical Address (IPA).

- Support of four different page sizes through two levels of translation tables using the short-descriptor translation table format as shown in figure 3.1:

  - Supersection: 16 MB memory region, translation properties are held in the first-level table.
  - Section: 1 MB memory region, translation properties are held in the first-level table.
  - Large page: 64 KB memory region, translation properties are held in the second-level table and the entries of the first-level tables contain translation properties and pointers to the second-level table.

- Small page: 4 KB memory region, translation properties are held in the second-level table and the entries of the first-level tables contain translation properties and pointers to the second-level table.

- Support of three different page sizes through three levels of translation tables using the long-descriptor translation table format as shown in figure 3.2:
  - 1 GB memory block
  - 2 MB memory block
  - 4 KB memory page

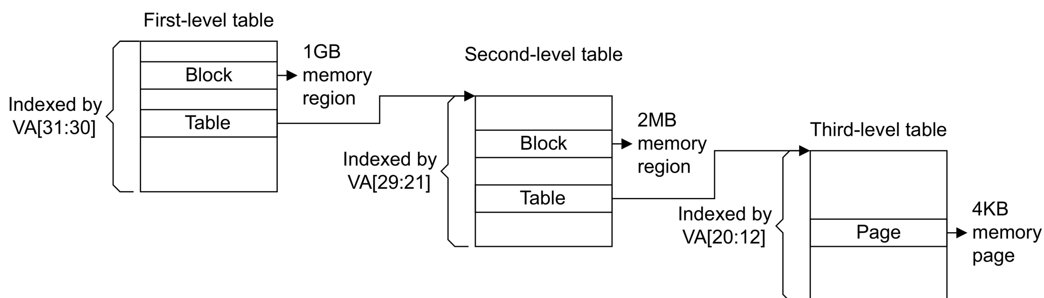

Figure 3.2: Translation table structure for stage 1 address translations using long-descriptor format. Taken from [ARM11, p. 1329], slightly modified to hide unnecessary details.

- 2-level TLB structure:
  - 32-entry fully-associative first level instruction TLB.
  - Two separate 32-entry fully associative first level TLBs for data load and store pipelines.
  - The first level TLBs provide a lookup of the virtual addresses in a single clock cycle.
  - Misses from the first level instruction and data TLBs are handled by a unified 4-way set-associative 512-entry second level TLB. Accesses to the second level TLB take a variable number of cycles, according to competing requests from each of the first level TLBs and other implementation-dependent factors. [ARM12a, p. 5-3]
  - No lockable TLB entries.

- Implements a hardware page table walk.

- The TLB entries contain a Virtual Machine Identifier (VMID) to permit virtual machine switches without TLB flushes.

- Global and address space identifiers, corresponding to PIDs.

- Additionally to the access permissions already described at ARM Cortex A9 in section 3.1.1, ARM Cortex A15 provides a Privileged Execute Never (PXN) bit to prohibit executing of instructions by system software from the corresponding memory regions.

3.2 Memory Management Units of the Analog Devices Inc. Blackfin Architecture represented by Analog Devices Inc. ADSP-BF54x

The complete list of the features of the MMU and explanations thereof can be found in [Ana10, p. 3-52 ff]. The relevant key features are provided in the following list:

- No support for virtual memory addressing.
- Two 16-entry associative memory blocks, where each entry is referred to as a Cacheability Protection Lookaside Buffer (CPLB) descriptor.
- The CPLB entries are divided in 16 instruction and 16 data CPLBs.
- In case that more than 32 CPLBs are needed, it is up to the OS to implement a page descriptor table that fulfills the requirements.
- Support of four different page sizes:
  - 1 KB
  - 4 KB
  - 1 MB
  - 4 MB
- Page properties [Ana10, p. 3-54]:
  - Cacheable: Determines if the page may be cached.
  - Dirty/modified: Marks a page as modified if the data was changed since the CPLB was last loaded.
  - Supervisor write access permission: Enables or disables writes by supervisor software to this page, only possible for data pages.
  - User write access permission: Enables or disables writes by user software to this page, only possible for data pages.
  - User read access permission: Enables or disables reads by user software from this page.
  - Valid: Marks the entry as valid.
  - Lock: Keep this CPLB entry. The actual CPLB replacement strategy is implemented by the OS.

3.3 Memory Management Units of the Power Architecture represented by Freescale Semiconductor e200z4

The complete list of the features of the MMU and explanations thereof can be found in [Fre09, p. 10-1 ff]. The relevant key features are provided in the following list:

- Supports an 8-bit PID.
- 16-entry fully associative TLB.
- The TLB is mostly software managed but the writing of new entries is hardware supported by some specialized registers and operations.
- Support of 23 page sizes (1 KB, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB).
- Freescale Semiconductor Book E Implementation Standards (EIS) MMU architecture compliant:
  - "*This MMU model defines MMU Assist (MAS) Special-Purpose Registers (SPRs), which hold the translation, memory protection, and other memory attributes necessary for page translation. It also defines that the tlbre and tlbwe instructions transfer this configuration information between the MAS registers and the TLBs.*" [Fre05, p. 18]
  - "*The TLB is accessed indirectly through several MAS registers. Software can write and read the MAS registers with mtspr and mfspr instructions*" .
- "*Support for external control of entry matching for a subset of Translation ID (TID) values to support non-intrusive runtime mapping modifications.*"
  - The TID field of an entry is compared with the PID of the virtual address that is to be translated. Entries can be marked global by a TID value of 0.
- Two virtual address spaces for instruction accesses and two virtual address spaces for data accesses, where one of each might be used for supervisor software respectively user software.
- Each of the following permissions has a corresponding bit in the page table entry:
  - Supervisor read permission
  - Supervisor write permission
  - Supervisor execute permission
  - User read permission
  - User write permission
  - User execute permission

3.4 Memory Management Units of the SPARC Architecture represented by Aeroflex Gaisler LEON3 and LEON4

The features of the MMU of LEON3 (see [Aer12, p. 694 ff]) and LEON4 (see [Aer12, p. 725 ff]) are the same, because both processors implement a SPARC V8 Reference MMU. The complete list of the features of the MMU and explanations thereof can be found in [SPA92, p. 241 ff]. The relevant key features are provided in the following list:

- Support of four different page sizes and sparse address spaces through three levels of translation tables. Note that Aeroflex Gaisler states that the page size is fixed at 4 KB and refers to the other page sizes as large linear mappings.
  - 4 GB
  - 16 MB
  - 256 KB
  - 4 KB
- Physical address space of up to 36 bit[8]
- Implements a hardware page table walk.
- The MMU can be configured to:
  - use a shared TLB or two separate TLBs for instruction and data entries.
  - hold between 2 and 32 fully associative entries in any TLB, which gives a maximum of 64 entries by using two separate TLB.
- Support for multiple contexts, where only one address space is active at any time. The mechanism relies on a context register in combination with a context table. The content of the context register is used as an index in the context table, which in turn points to the first level page table. Thus no TLB flush is needed on a context switch but there are no global entries possible.
- It is possible to store a page table entry at any translation table level to implement different page sizes. Furthermore, the first level(s) must hold a pointer to the next level, termed page table descriptor, in case that the page table entry is stored in one of the following levels. There are three possible entry types, which are marked by a two bit field in every table entry: page table descriptor, page table entry and invalid entry.
- The properties of a page table entry are as follows:

[8] An in-depth analysis of the VHDL source code of LEON3, as provided by Aeroflex Gaisler in its GRLIB, revealed that the actual AMBA address space is only 32 bit. Hence, the top most bits of the 36 bit physical page address are discarded within the processor. Aeroflex Gaisler has confirmed this behavior after consultation by e-mail. The regarding correspondence is quoted in appendix A.

Accesses allowed	
User Access	Supervisor Access
Read Only	Read Only
Read/Write	Read/Write
Read/Execute	Read/Execute
Read/Write/Execute	Read/Write/Execute
Execute Only	Execute Only
Read Only	Read/Write
No Access	Read/Execute
No Access	Read/Write/Execute

Table 3.2: Access permissions for user and supervisor software at page table entries of Aeroflex Gaisler LEON 3 and LEON 4

- Cacheable: This bit selects if the page may be cached.
- Modified: This bit is set by the MMU on a write access to the page.
- Referenced: This bit is set by the MMU whenever the page is accessed.
- The possible access permissions for user and supervisor software are listed in table 3.2.

3.5 Memory Management Unit of the Altera Corporation NIOS II Architecture

See [Alt11, p. 2-15 ff]

- 2-level TLB structure:
  - Two separate first level TLBs, termed micro TLBs, stored in Logic Element (LE)-based registers for instructions and data. Those are fully associative and have a configurable number of entries. Default values are six entries for data and four entries for instructions.
  - One second level TLB, termed main TLB, stored in on-chip RAM for both instructions and data. This is n-way set associative, where the number of ways and entries are both configurable by processor parameters. The default value is a 16-way set associativity and the default number of entries depends on the devices, e.g., Altera Corporation Cyclone II, Stratix II and Stratix II GX default to 128 entries, whereas Altera Corporation Cyclone III, Stratix III and Stratix IV default to 256 entries.
  - On a TLB lookup the processor first searches the micro TLB, in case of a miss it performs a lookup in the main TLB. If this is a hit, the entry is copied to the corresponding micro TLB. The replacement strategy in the micro TLBs is Least Recently Used (LRU) and performed in hardware. If the lookup in main TLB also misses, the OS is invoked.

Partition	Virtual Address Range	Used by	Memory Access	User Mode Access	Default Data Cacheability
I/O	0xE0000000 – 0xFFFFFFFF	OS	Bypasses TLB	No	Disabled
Kernel	0xC0000000 - 0xDFFFFFFF	OS	Bypasses TLB	No	Enabled
Kernel MMU	0x80000000 - 0xBFFFFFFF	OS	Uses TLB	No	Set by TLB
User	0x00000000 - 0x7FFFFFFF	User processes	Uses TLB	Set by TLB	Set by TLB

Table 3.3: Virtual memory partitions of Altera Corporation NIOS II processors, based on [Alt11, p. 3-4 f]

- The 4 GB virtual address space is separated in partitions as shown in table 3.3. Hence, 2 GB are reserved for the OS and 2 GB are reserved for user processes.

- The physical memory is divided in two memory regions, the lowest 512 MB are termed low memory and the rest is called high memory, as shown in figure 3.3. "*Any physical address in low memory (29-bits or less) can be accessed through the TLB or by bypassing the TLB. When bypassing the TLB, a 29-bit physical address is computed by clearing the top three bits of the 32-bit virtual address.*" [Alt11, p. 3-5]

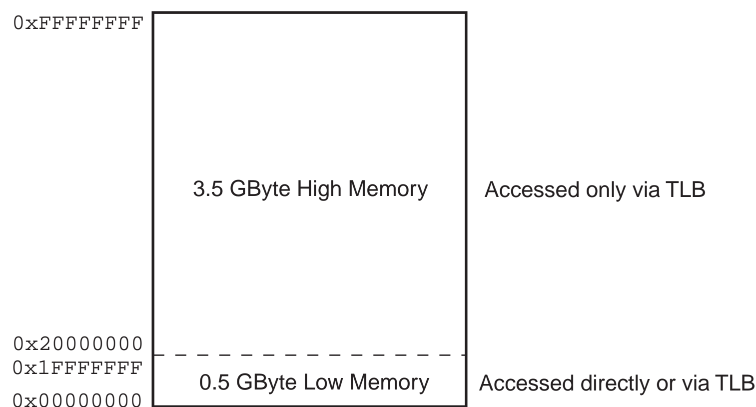

Figure 3.3: Division of physical memory of Altera Corporation NIOS II processors, taken from [Alt11, p. 3-5].

- Each TLB entry holds a PID and a global flag. In case the later is set, the former is ignored.
- The following properties can be set on a per page basis:
  - Cacheable
  - Readable
  - Writable
  - Executable
  - "*Because there is no "valid bit" in the TLB entry, the operating system software invalidates the TLB by writing unique VPN* [Virtual Page Number] values from the I/O partition of virtual addresses into each TLB entry." [Alt11, p. 3-7]
- Fixed 4 KB page and page frame size.
- The TLB acts as a software managed cache for page table entries. Thus the actual page table structure, replacement policy and page table walks are up to the OS.
- Note that this processor can also be configured to use a MPU, though the MMU and MPU are mutually exclusive [Alt11, p. 2-16].

CHAPTER 4

Failure Mode Analysis of a Hypothetical Memory Management Unit

The basis of this chapter is the partitioning of a hypothetical MMU in multiple function blocks as shown in figure 4.1. This MMU contains some optional features that are shown as gray blocks in the figure and may be implemented in software instead of specialized hardware, without a huge performance penalty. Each of the blocks is going to be described with an emphasis on high level functionality and input/output signal vectors. Furthermore, an exhaustive high-level failure effects analysis is performed on every block with the assumption of all optional blocks also implemented in hardware. Finally, the causal chains of failure effects that lead to output failures of the MMU are examined.

4.1 Fault Hypothesis

In order to keep the complexity at a reasonable level the following discussion relies on a single-fault assumption w.r.t. faults within the MMU. To make the discussion more complete, also combinations of a fault at the MMU with a wrong memory access by the Central Processing Unit (CPU), e.g., due to a software fault, are considered. Most of the failure effects do also apply to permanent faults but the main focus lies on transient faults.

Due to the single-fault assumption, a delay fault may only affect a single signal line and therefore typically results, at the instant of sampling, either in corrupted data, a delayed control signal or different input perception on multiple blocks.

4.2 Description of Function Blocks

The MMU is typically interfaced either by the CPU or the cache system, thus the Request (REQ) and Response (RES) signal vectors are connected with one of those units. The blocks that are marked as optional are only needed in case of a hardware implemented table walk and could thus be replaced by software.

The acronyms in parentheses describe the origin/destination block of the input/output signal vectors at the inputs and outputs sections of the following function block descriptions. For example the entry "(TM) Virtual page number" at the output description of the Virtual Page Number Extraction (VPNE) block means that the signal vector for the virtual page number leads from the TLB Manager (TM) to the VPNE block.

Failure effects that are categorized by a **T-** influence only the timing behavior of the MMU and thus are relevant for real-time systems but are not examined in detail. Failure effects categorized by **F-** may have severe consequences for the active process (**FA-**) or may even influence the memory of another process (**FO-**). The categorizations **FCA-**, **FCO-** and **TC-** have in principle the same meaning, but represent types where the failure occurs due to a combination of multiple blocks.

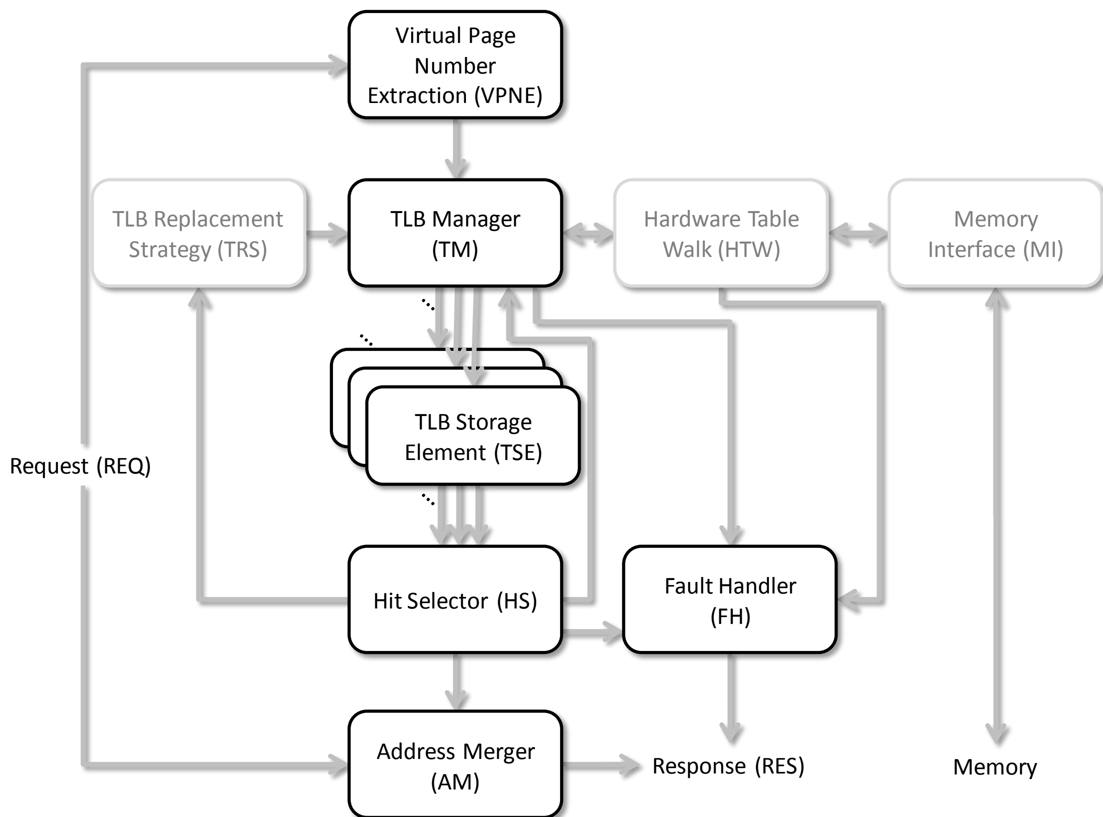

Figure 4.1: Function blocks of a MMU

4.2.1 Considered MMU Functions

The hypothetical MMU features a hardware implemented table walk in combination with a hardware implemented TLB entry replacement strategy. Furthermore, it supports multiple page sizes through multilevel page tables.

4.2.1.1 Page Table Entry

In the following discussion a Page Table Entry (PTE) and thus a TLB entry of the hypothetical MMU is assumed to contain the following information:

- Entry type: In case of a multilevel page table this can either be a PTE or a Page Table Descriptor (PTD), where the latter means that it is a pointer to another page table. Note that the TLB typically only contains PTEs, because PTDs are not sufficient to access physical memory.
- Access permissions of the page.
- Modified indicator that gets set whenever the memory access using the page is a write access.
- Referenced indicator that gets set whenever the page is accessed.
- Valid indicator that is cleared whenever the entry is flushed from the TLB or swapped out.
- Virtual page number, possibly stored in multiple entries to support multilevel page tables.
- In case of a multilevel page table: Level at which the entry hits.
- PID of the page.
- Physical page frame number of the page.

4.2.2 Virtual Page Number Extraction (VPNE)

- Intended function:

  This block extracts the page address from the virtual address. If the MMU supports multilevel page tables, the virtual address is split up in separate entries for every level (see section 2.2.5). Note that, in case of multiple page sizes, this function block can always separate the virtual address in all separate levels. The parts that are not needed may simply be neglected at the following blocks.

  For simplicity every other input signal vector of the MMU, besides the offset vector, is considered to be passed through this block.
- Inputs:
  - (REQ) Virtual address

  - (REQ) Operation to perform (read entry, flush entry, in case of no hardware table walk: write entry)
  - (REQ) Various control data, e.g., current PID and page table base pointer
  - (REQ) Supervisor access indicator
  - (REQ) Read access or write access to memory
  - In case of no hardware table walk: (REQ) New TLB entry for entry write operations
  - In case of no hardware table walk: (REQ) Current replacement position for entry write operations

- Outputs:
  - (TM) Virtual page number
  - (TM) Operation to perform (read entry, flush entry, in case of no hardware table walk: write entry)
  - (TM) Various control data, e.g., current PID and page table base pointer
  - (TM) Supervisor access indicator
  - (TM) Read access or write access to memory
  - In case of no hardware table walk: (TM) New TLB entry for entry write operations
  - In case of no hardware table walk: (TM) Current replacement position for entry write operations

- Stored data:
  - None

- Example implementation: The whole block may be implemented as simple signal vectors.

- Potential failure effects:

FA-VPNE 1 The virtual address is corrupted.

FO-VPNE 2 The data that is passed through is corrupted.

4.2.3 Optional: TLB Replacement Strategy (TRS)

- Intended function:

  This block determines which TLB entry is to be replaced and is needed if the TLB entries are managed by hardware. Possible replacement strategies are, e.g., LRU or pseudo-random. Whenever the calculated entry is marked as consumed, the block has to calculate a new one. In LRU strategies this can also happen implicitly because if the entry is used, it is written and thus can no longer be the least recently used one. The TRS can be implemented to mark a recently flushed entry as the least recently used one.

- Inputs:
  - (TM) TLB entry consumed indicator
  - (TM) reset TRS
  - In case of a hardware implemented LRU strategy: (HS) TLB entry flushed indicator
  - In case of a hardware implemented LRU strategy: (HS) TLB entry touched indicator
  - In case of a hardware implemented LRU strategy: (HS) Position of the entry that was flushed or touched
- Outputs:
  - (TM) Current replacement position
- Stored data:
  - In case of a hardware implemented LRU strategy: The order in which the entries were recently accessed.
- Example implementation: Pseudo-random can be, e.g., implemented as a simple counter that is incremented on every page table walk until it wraps. LRU is a complex hardware block.
- Potential failure effects:

T-TRS 1 The entry, chosen for replacement, is another one than it should be according to the replacement strategy.

FO-TRS 2 The chosen entry is changed during a write access to the TSE.

4.2.4 TLB Storage Element (TSE)

- Intended function:

  For a fully associative TLB, the TLB storage element may be implemented as a CAM that stores one entry. If the TLB is set-associative, the TSE is directly accesses by the TM and contains a number of CAMs that depends on the level of associativity. At direct mapped TLBs no CAMs are involved.

  In case that the input virtual page number and the PID match an entry the storage element needs to output a hit indicator and the entry. The storage element need not provide any further fault handling.

  In case the memory access is a write access, the stored entry gets marked as modified. Whenever the entry is accessed its referenced indicator is asserted. Furthermore, if a stored entry gets modified somehow, a synchronize indicator is asserted.

- Inputs:

- (TM) Virtual page number
- (TM) Operation to perform (read entry, write entry, flush entry)
- (TM) PID
- (TM) Supervisor access indicator
- (TM) Read access or write access to memory
- (TM) New TLB entry for entry write operations

- Outputs:
  - (HS) TLB entry
  - (HS) Hit indicator
  - In case of a hardware implemented LRU strategy: (HS) TLB entry touched indicator
  - In case of a hardware implemented LRU strategy: (HS) TLB entry flushed indicator
  - In case of a hardware implemented LRU strategy: (HS) Synchronize request indicator

- Stored data:
  - All the data of its TLB entries (see section 4.2.1.1).
  - Typically at least one state variable.

- Example implementation: Contains some type of memory element, the remainder is a complex hardware block.

- Potential failure effects:

FO-TSE 1 The stored data is corrupted.

T-TSE 2 No hit is indicated although there is a matching entry.

FO-TSE 3 A wrong but valid entry is output and a hit indicated.

FO-TSE 4 A corrupted entry is output and a hit is indicated.

FO-TSE 5 A corrupted entry is output and a synchronize request is indicated.

FO-TSE 6 A wrong but valid entry is output and a synchronize request is indicated.

FA-TSE 7 No synchronize request is indicated although necessary.

T-TSE 8 The entry gets unintendedly flushed.

FO-TSE 9 The entry stays valid although it should be flushed.

FO-TSE 10 The PID check passes although it should not.

T-TSE 11 The PID check does not pass although it should.

FO-TSE 12 The returned TLB entry is outdated.

4.2.5 Hit Selector (HS)

- Intended function:

  This block selects the entry of the TLB storage element that has its hit indicator asserted and also asserts its own hit indicator. In case that no storage element has its hit indicator asserted, it clears its own hit indicator. The same is true for the synchronize request. Hence, the output hit indicator, in its simplest form, is a logical or over all input hit indicators and the output synchronize request is a logical or over all synchronize requests. However, only one of the inputs may be asserted at a time, which could also be checked by this block.

  In both cases, either asserted hit indicator or synchronize request, the corresponding TLB entry is output.

- Inputs:
  - (TSE) Multiple TLB entries
  - (TSE) Multiple hit indicators
  - (TSE) Multiple synchronize requests
  - In case of a hardware implemented LRU strategy: (TSE) Multiple TLB entry touched indicators
  - In case of a hardware implemented LRU strategy: (TSE) Multiple TLB entry flushed indicators
  - In case of a hardware implemented LRU strategy: (TSE) Multiple Synchronize request indicator

- Outputs:
  - (AM+FH+TM) TLB Entry
  - (TM+FH) Hit indicator
  - (TM) Synchronize request indicator
  - In case of a hardware implemented LRU strategy: (TRS) TLB entry flushed indicator
  - In case of a hardware implemented LRU strategy: (TRS) TLB entry touched indicator
  - In case of a hardware implemented LRU strategy: (TRS) Position of the entry that was flushed or touched

- Stored data:
  - None

- Example implementation: May be implemented as a multiplexer.

- Potential failure effects:

T-HS 1 No hit is indicated and no entry output although there is a matching entry.

T-HS 2 A synchronize request is indicated although not necessary.

FA-HS 3 No synchronize request is indicated although necessary.

FO-HS 4 A synchronize request is indicated and a wrong but valid entry is output.

FO-HS 5 A synchronize request is indicated and a corrupted entry is output.

FO-HS 6 A hit is indicated and a wrong but valid entry is output.

FO-HS 7 A hit is indicated and the output TLB entry is corrupted.

T-HS 8 In case of a hardware implemented LRU strategy: A wrong position of the entry that was flushed or touched is output.

T-HS 9 In case of a hardware implemented LRU strategy: No flush is indicated although there was one.

T-HS 10 In case of a hardware implemented LRU strategy: No touch is indicated although there was one.

T-HS 11 In case of a hardware implemented LRU strategy: A flush is indicated although there was none.

T-HS 12 In case of a hardware implemented LRU strategy: A touch is indicated although there was none.

4.2.6 Optional: Hardware Table Walk (HTW)

- Intended function:

  This block receives PTDs for each page table level from the memory hierarchy until a PTE is fetched. The starting point is determined by the physical base address of the first level page table. If there is a fault like, e.g., a memory access fault or a page fault, a corresponding fault indicator is asserted. During a normal page table walk the resulting entry is sent to the TM block. In case the table walk was started due to a synchronize request, the entry is searched and updated accordingly in memory.

- Inputs:
  - (TM) Virtual page number
  - (TM) Various control data like, e.g., current PID and page table base pointer
  - (TM) Table walk request
  - (TM) Synchronize request
  - (TM) TLB entry to synchronize
  - (MI) Ready indicator
  - (MI) Data from memory

- Outputs:

  - (TM) TLB entry
  - (FH+TM) Fault status vector
  - (MI) Data to memory
  - (MI) Memory address
  - (MI) Read or write indicator

- Stored data:
  - Page table base pointer
  - State variable

- Example implementation: Typically is a complex hardware block.

- Potential failure effects:

T-HTW 1 Indicates a page fault although a page exists.

FO-HTW 2 Does not indicate a page fault although no page exists.

FO-HTW 3 Returns a wrong but valid TLB entry.

FO-HTW 4 Returns corrupted data.

FO-HTW 5 Writes a wrong but valid entry back to memory.

T-HTW 6 Does not fail to access memory but indicates a memory access fault.

T-HTW 7 Fails to access memory but does not indicate a corresponding fault.

FO-HTW 8 Accesses wrong memory address.

FO-HTW 9 Writes corrupted data to memory.

FA-HTW 10 Does not perform a synchronization although requested.

T-HTW 11 Does not perform a table walk although requested.

T-HTW 12 Does perform a synchronization although not requested.

T-HTW 13 Does perform a table walk although not requested.

FO-HTW 14 Performs a synchronization although a table walk was requested.

FA-HTW 15 Performs a table walk although a synchronization was requested.

4.2.7 TLB Manager (TM)

- Intended function:

  In case of a fully associative TLB structure, the TM block broadcasts the request from the VPNE to every TSE. Whereas at a set-associative TLB the TSE that is to be accesses is determined by the virtual page number.

  If the MMU features a hardware implemented page table walk this block provides some management functions for that. The block invokes a hardware table walk if there is a TLB

miss and stores the received TLB entry at the storage element indicated by the replacement strategy, if the page table walk was successful. In case of a synchronize request of the HS, the TLB manager issues an update of the page entry in memory by the table walk module.

If there is no hardware implemented page table walk, a new entry needs to be supplied by the REQ to the MMU, which is in turn passed on by the VPNE. Furthermore, in case that the TRS is also implemented in software, the replacement position also has to be provided.

- Inputs:
  - (VPNE) Virtual page number
  - (VPNE) Operation to perform (read entry, flush entry, in case of no hardware table walk: write entry)
  - (VPNE) Supervisor access indicator
  - (VPNE) Various control data like, e.g., current PID and page table base pointer
  - (VPNE) Read access or write access to memory
  - (HS) Hit indicator
  - (HS) Synchronize request indicator
  - (HS) TLB entry to synchronize
  - (HTW or VPNE) New TLB entry
  - (TRS or VPNE) Current replacement position
  - In case of a hardware implemented table walk: (HTW) Fault status vector
- Outputs:
  - (TSE+ optionally: HTW) Virtual page number
  - (TSE) Operation to perform (read entry, write entry, flush entry)
  - (TSE+ optionally: HTW) Various control data like, e.g., current PID and page table base pointer
  - (TSE+FH) Supervisor access indicator
  - (TSE) Read access or write access to memory
  - (TSE) New TLB entry
  - (TRS) TLB entry consumed indicator
  - (TRS) reset TRS
  - In case of a hardware implemented table walk: (HTW) Table walk request
  - In case of a hardware implemented table walk: (HTW) TLB entry to synchronize.
  - In case of a hardware implemented table walk: (HTW) Synchronize request
- Stored data:

- None

- Example implementation: Typically is a complex hardware block.
- Potential failure effects:

T-TM 1 Stores every TLB entry at the same TSE.

FA-TM 2 Indicates a write access from CPU to memory, although it is a read access.

FA-TM 3 Indicates a read access from CPU to memory, although it is a write access.

FA-TM 4 Indicates a different operation than requested.

FO-TM 5 Outputs a wrong PID.

FA-TM 6 Indicates a supervisor access although it is none.

FA-TM 7 Indicates no supervisor access although it is one.

T-TM 8 Outputs no TLB entry consumed although a entry was stored at the position.

T-TM 9 Outputs a TLB entry consumed although no entry was stored at the position.

T-TM 10 Resets the TRS permanently.

T-TM 11 Resets the TRS randomly.

FA-TM 12 Outputs a wrong virtual page number.

FO-TM 13 Passes on a corrupted TLB entry.

FO-TM 14 Passes on a wrong but valid TLB entry.

FO-TM 15 In case of a hardware implemented table walk: Outputs a wrong page table base pointer.

FA-TM 16 In case of a hardware implemented table walk: Invokes a table walk for a wrong virtual address.

T-TM 17 In case of a hardware implemented table walk: Invokes a table walk although not needed.

T-TM 18 In case of a hardware implemented table walk: Invokes no table walk although needed.

T-TM 19 In case of a hardware implemented table walk: Invokes a synchronization although no request was received.

FA-TM 20 In case of a hardware implemented table walk: Invokes no synchronization although needed.

FO-TM 21 In case of a hardware implemented table walk: Passes on a synchronize request with a corrupted TLB entry.

FO-TM 22 In case of a hardware implemented table walk: Passes on a synchronize request with a wrong but valid TLB entry.

T-TM 23 In case of a hardware implemented table walk: Does not store a TLB entry although a page table entry was received from the HTW module.

4.2.8 Fault Handler (FH)

- Intended function:

  This block checks if the received TLB entry allows the requested memory access and stores the result in a fault vector. Furthermore, the fault vector of the HTW is combined with the fault vector of this block. The resulting vector is part of the RES from the MMU and can be used to raise an exception and inhibit memory accesses.

- Inputs:

  - (HS) TLB Entry
  - (HS) Hit indicator
  - (TM) Supervisor access indicator
  - (HS) Read access or write access to memory
  - In case of a hardware implemented table walk: (HTW) Fault status vector

- Outputs:

  - (RES) Fault status vector

- Stored data:

  - none

- Example implementation: Contains various gates, bus lines and comparators.

- Potential failure effects:

T-FH 1 Indicates a fault although there should have been none (cases depend on supported faults).

FO-FH 2 Indicates no fault although there should have been one (cases depend on supported faults).

T-FH 3 Indicates wrong fault (cases depend on supported faults).

4.2.9 Address Merger (AM)

- Intended function:

  This block merges the resulting page frame address with the offset that can be obtained from the virtual address of the REQ to the MMU. In case of a multilevel page table, the level at which the entry hits determines the partition of the physical address in an offset and a page frame part.

- Inputs:

  - (REQ) Virtual address offset

- (HS) TLB entry

- Outputs:
  - (RES) Physical address
- Stored data:
  - None
- Example implementation: May be implemented as simple bus lines. In case of a multilevel page table it also has to contain gates and multiplexers.
- Potential failure effects:

FO-AM 1 Merges page frame address and offset at the wrong position.

FO-AM 2 Outputs corrupted page frame address and offset.

FO-AM 3 Outputs corrupted page frame address.

FA-AM 4 Outputs corrupted offset.

FO-AM 5 Outputs a wrong but valid page frame address.

FA-AM 6 Outputs a wrong but valid offset.

4.2.10 Optional: Memory Interface (MI)

- Intended function:

  The MI block provides an interface to the memory hierarchy and is needed to supply the optional hardware page table walk with the necessary data and to allow updating of PTEs due to synchronize request.
- Inputs:
  - (HTW) Data to memory
  - (HTW) Memory address
  - (HTW) Read or write indicator
  - (MEM) Data from low level memory interface
- Outputs:
  - (HTW) Ready indicator
  - (HTW) Data
  - (MEM) Low level memory interface
- Stored data:

  - State variable
  - Address and data to be stored

- Example implementation: Can be implemented as a AMBA High-performance Bus (AHB) interface.

- Potential failure effects:

FO-MI 1 Writes corrupted data to memory.

FO-MI 2 Writes wrong but valid data to memory.

FO-MI 3 Writes data to wrong but valid address.

FO-MI 4 Reads wrong but valid address.

T-MI 5 Writes data to invalid address.

T-MI 6 Reads invalid address.

FO-MI 7 Outputs corrupted data to HTW.

FO-MI 8 Outputs wrong but valid data to HTW.

FO-MI 9 Indicates ready although not ready.

T-MI 10 Fails to set ready indicator although ready.

4.3 Failure Effects Involving Multiple Function Blocks

The hypothetical MMU contains some signal vectors that are inputs of multiple function blocks. Thus, the possibility of different input perception arises. Due to the complexity of this type of failure effect, each of the possible cases is going to be analyzed in detail. Note that an actual implementation of a MMU may contain more or different such forks.

4.3.1 Fault Vector from HTW to FH and TM

This signal vector allows the TM to determine whether the invoked hardware table walk was successfully or not and is combined with the result of the permission checking by the FH block.

TC-HTW 1 If the table walk was successfully but the TM block erroneously reads a fault indicator, then the TM does not take over the entry from the HTW and hence exhibits T-TM 18. Nevertheless, the FH does not output a fault indication. Due to the fact that the FH also does not read a hit indication, no address translation occurs. If properly handled, this kind of failure should only delay a translation or result in no translation but does not cause a more severe wrong translation. In case the actual implementation of the FH block outputs a corresponding fault indication this type of failure effect also results in T-FH 1.

TC-HTW 2 If the table walk was successfully but the FH block erroneously reads a fault indicator, the TM block stores a correct entry at a TSE but the FH outputs a fault indication. Thus, the correct translation is handled like a fault corresponding to the output of the FH block and results in no wrong translation. This type of failure effect equates T-FH 1.

FCO-HTW 3 If the table walk was not successfully but the TM block erroneously reads no fault indicator, the TM may store a corrupted entry at a TSE and hence exhibits FO-TM 13. The FH still outputs the fault indicated by the HTW and thus at first no wrong translation occurs. Nevertheless, there is a possibility that the stored TLB entry later on matches an incoming translation request and thus leads to a wrong translation.

TC-HTW 4 If the table walk was not successfully but the FH block erroneously reads no fault indicator, then the TM does not take over the entry from the HTW. Nevertheless, the FH does not output a fault indication. Due to the fact that the FH also does not read a hit indication, no address translation occurs. If properly handled, this kind of failure should only delay a translation or result in no translation but does not cause a more severe wrong translation. In case the actual implementation of the FH block outputs a corresponding fault indication – and that indication is considered a wrong indication – this type of failure effect equates to T-FH 3.

4.3.2 TLB Entry from HS to AM, FH and TM

FCO-HS 1 If a TLB entry is differently perceived between those three blocks, at least one of them reads a wrong or corrupted entry. Hence, the corresponding blocks show misbehavior. Due to the wide influence of the TLB entry, there are many failure effect possibilities but all of them are already covered by the discussion of the single blocks. The corresponding failure effects are FO-AM 1, FO-AM 3, FO-AM 5, T-FH 1, FO-FH 2, T-FH 3, FO-TM 21 and FO-TM 22.

4.3.3 Hit indicator from HS to TM and FH

The hit indicator signal is used to determine if the TM block has to invoke a hardware table walk and if the FH block may use the TLB entry to check the permissions for the current translation.

TC-HS 2 If the HS indicates a hit but the TM reads no hit indication, it invokes a hardware table walk, which shows up as failure effect T-TM 17. Nevertheless, the FH checks the permissions of the corresponding TLB entry. Thus, the HS block has to ensure that the TLB entry at its output stays valid during the whole translation, although a table walk for the same page table entry is in progress. If this is the case, no wrong translation occurs.

FCO-HS 3 If the HS indicates a hit but the FH reads no hit indication, it does not perform a permission check. Hence, there is a possibility that the fault indication by the FH block shows the failure effects T-FH 1, FO-FH 2 and T-FH 3.

TC-HS 4 If the HS indicates no hit but the TM reads a hit indication and a non-hitting translation was in progress, the TM does not invoke a table walk although necessary (T-TM 18). Due to the no hit indication, no wrong translation occurs. If properly handled, this kind of failure only delays a translation or results in no translation.

FCO-HS 5 If the HS indicates no hit but the FH reads a hit indication, the FH checks the permissions of a wrong or corrupted entry and outputs the result. Hence, the fault indication by the FH block shows the failure effect T-FH 3, FO-FH 2 or T-FH 1.

4.3.4 Virtual Page Number from TM to TSE and HTW

The TM either accesses a TSE or invokes a hardware table walk or a synchronization by the HTW. Hence, those different inputs are not relevant at the same time and different input perception plays no role. The failure effects are the same as in the discussion of the single blocks.

4.3.5 Various Control Data from TM to TSE and HTW

The same argumentation as in section 4.3.4 applies to this signal vector.

4.3.6 Supervisor Access Indicator from TM to TSE and FH

The supervisor access indicator is either used to flush an entry from the TSE or for permission checking at the FH block. Hence, in principle the same argumentation as in section 4.3.4 also applies to this indicator.

4.4 Potential Failure Propagation

The following diagrams show causal chains of failure effects that lead to the effects on top of every diagram. Every single effect can either occur on its own, e.g., by a bit flip, or can be caused by another failure effect that has an arrow to it. The diagrams rely on a single failure assumption, thus no combination of multiple failures is considered.

Some diagrams contain sub-diagrams to lower the complexity. This is indicated by a filled rectangle. The matching sub-diagram has the same rectangle on its top level. Failure effects that lead to an output failure are indicated by a filled circle.

4.4.1 Output Failures after AM Block

Every failure that leads to a wrong physical address may have severe consequences. Whenever a process writes or reads a wrong memory location, there is a risk that the memory separation between the processes is violated. Thus, a failure of this type has the potential to influence the OS and cause a complete system failure.

There are three major causes of this type of failure: Either the offset portion of the virtual address received from the REQ gets corrupted before or within the AM block (see figure 4.2), or a wrong but valid (see failure FO-HS 6) or corrupted (see failure FO-HS 7) TLB entry is used to obtain the page frame address (see figure 4.3).

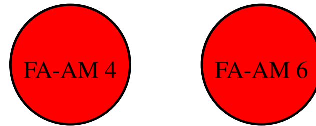

Figure 4.2: Failure effects related to the offset portion of the physical address have no predecessor failure effects within the MMU.

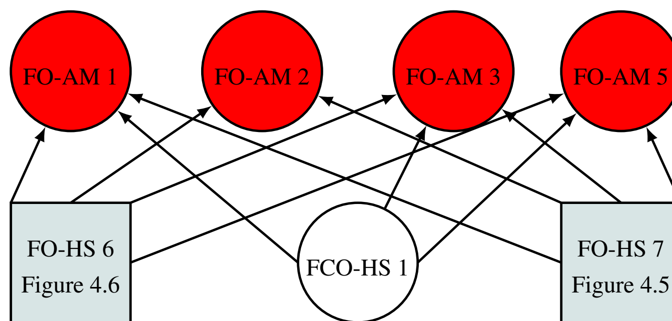

Figure 4.3: Failure effects that lead to a wrong page frame address at the AM block.

4.4.2 Output Failures after FH Block

Failure effects at the FH block may be severe if they occur together with a software failure, e.g., access of a prohibited virtual address. Although this type of error combination violates the single error assumption, it also has to be considered. It results in violation of the memory separation and could be either intentionally or unintentionally used to corrupt the OS.

There are three potential failure effects at the FH block: It can either indicate no fault although there should have been one (see figure 4.4), indicate a fault although there should have been none or indicate a wrong fault. The first effect may be severe if combined with an access to a wrong virtual address by software. The latter two result in no memory access and thus, if they occur transiently and not permanently, only influence the timing behavior and can be handled by the OS.

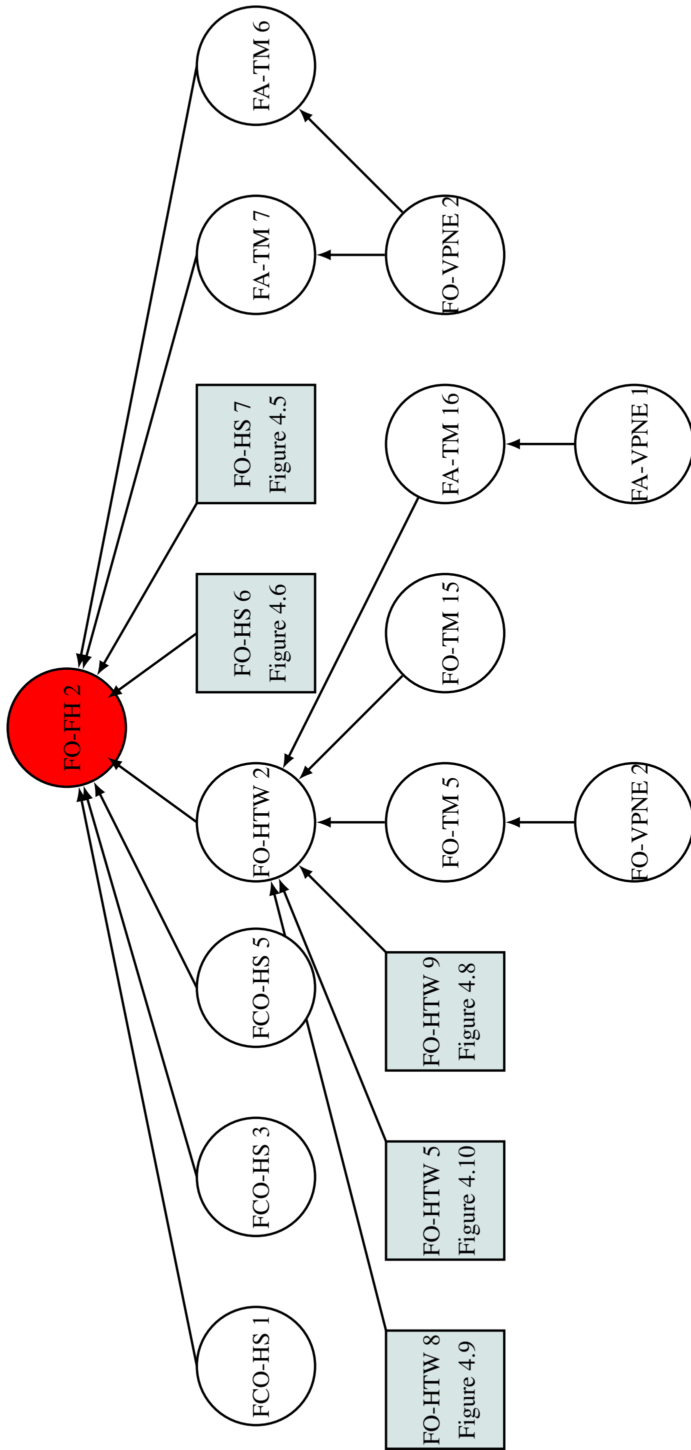

Figure 4.4: Failure effects that lead to the FH block indicating no fault although there should be one.

4.4.3 Output Failures after MI block

Due to the fact that the page table resides in physical address space, it has to be possible for the MI block to directly access main memory. Thus, if the MI block writes data to a wrong but valid memory address it may corrupt arbitrary data in memory and can thus lead to a complete system failure. Hence, this type of failure should also be considered an output failure.

A read from a wrong but valid address results either in a wrong but valid TLB entry (FO-MI 8) or more likely in corrupted data to the HTW (FO-MI 7). Both types are already covered and as they occur at an input, cannot be considered an output failure.

In case the memory address is an invalid address, i.e., an address that exceeds the physically addressable memory range, it is expected that the memory system either does not respond to the request or raises a corresponding error flag. Thus, no synchronization or table walk is performed and this results in a timing failure.

Hence, we can conclude that there is one critical output failure after the MI block, i.e., FO-MI 3. It has two possible consequences: Either it corrupts another page table entry, then it can be considered an intermediate failure or it corrupts arbitrary data in memory, then it has to be considered an output failure. Thus, FO-MI 3 is marked as an output failure but contained as an intermediate failure in figure 4.6 and figure 4.7.

4.4.4 Intermediate Failure Effects

To keep the complexity of the diagrams for output failures at manageable scale, the following sub diagrams are used.

Note that FO-HTW 4 (see figure 4.7) may be caused by FO-TM 21 (see figure 4.11), which in turn can be caused by FO-TSE 1 (see figure 4.13) and hence could again be caused by FO-HTW 4. This circular dependency should not be followed, the intention is simply that FO-TSE 1 may be caused by FO-HTW 4 and that FO-HTW 4 may also be caused by FO-TSE 1.

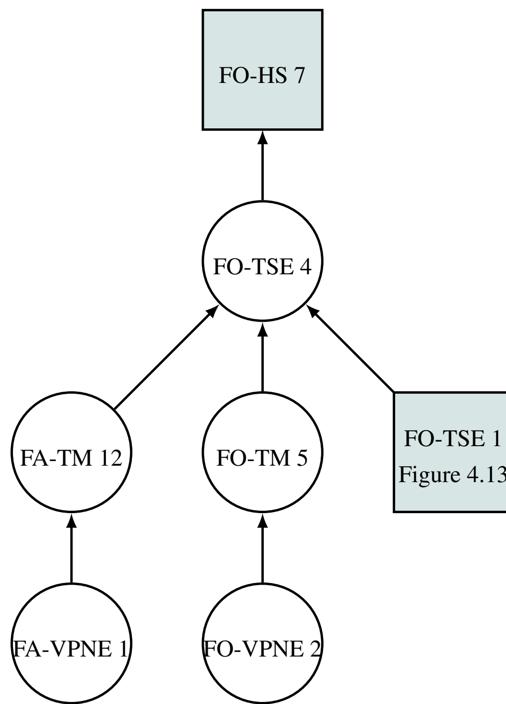

Figure 4.5: Failure effects that lead to the output of a corrupted TLB entry with an indicated hit from the HS block.

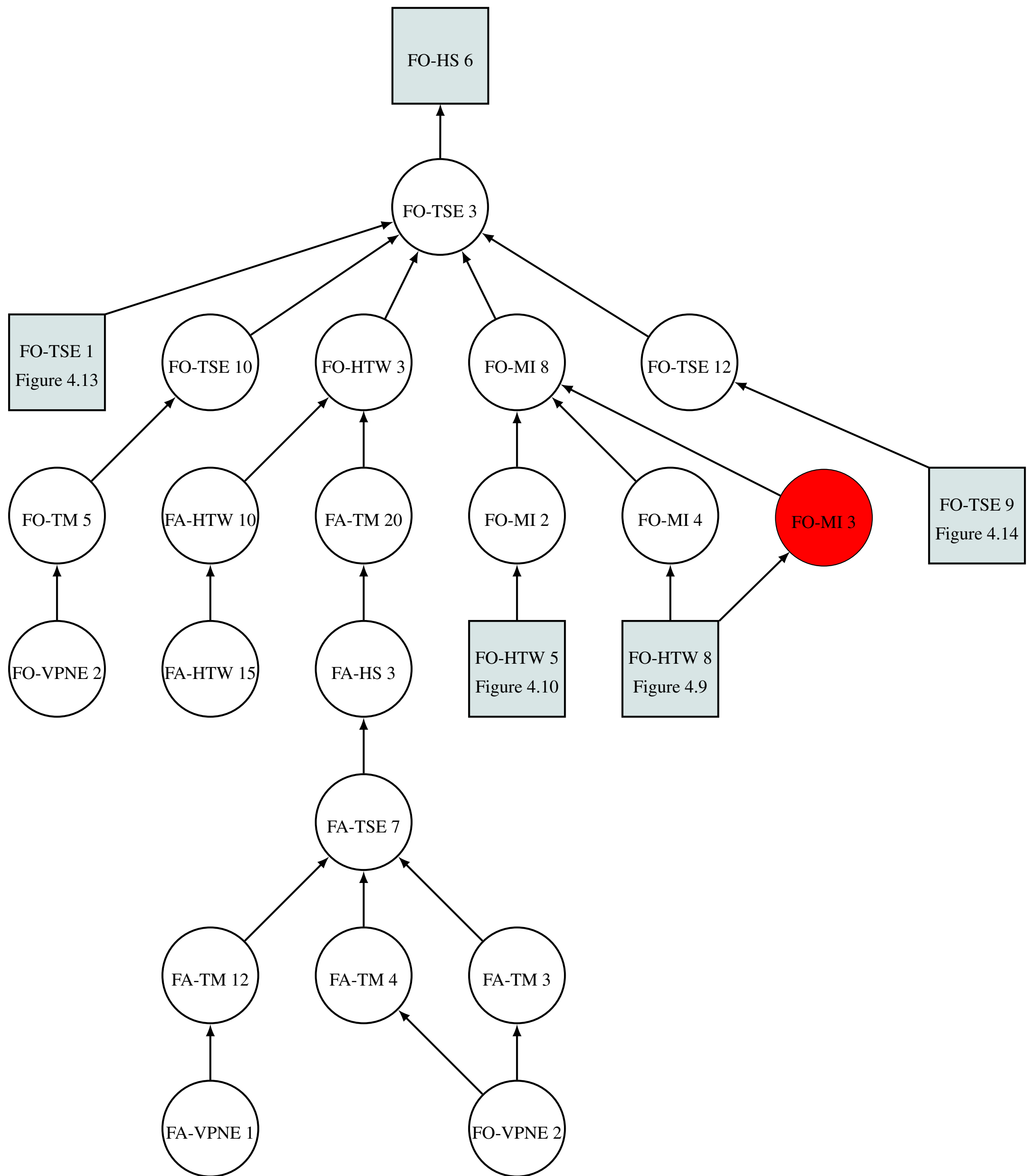

Figure 4.6: Failure effects that lead to the output of a wrong but valid TLB entry with an indicated hit from the HS block.

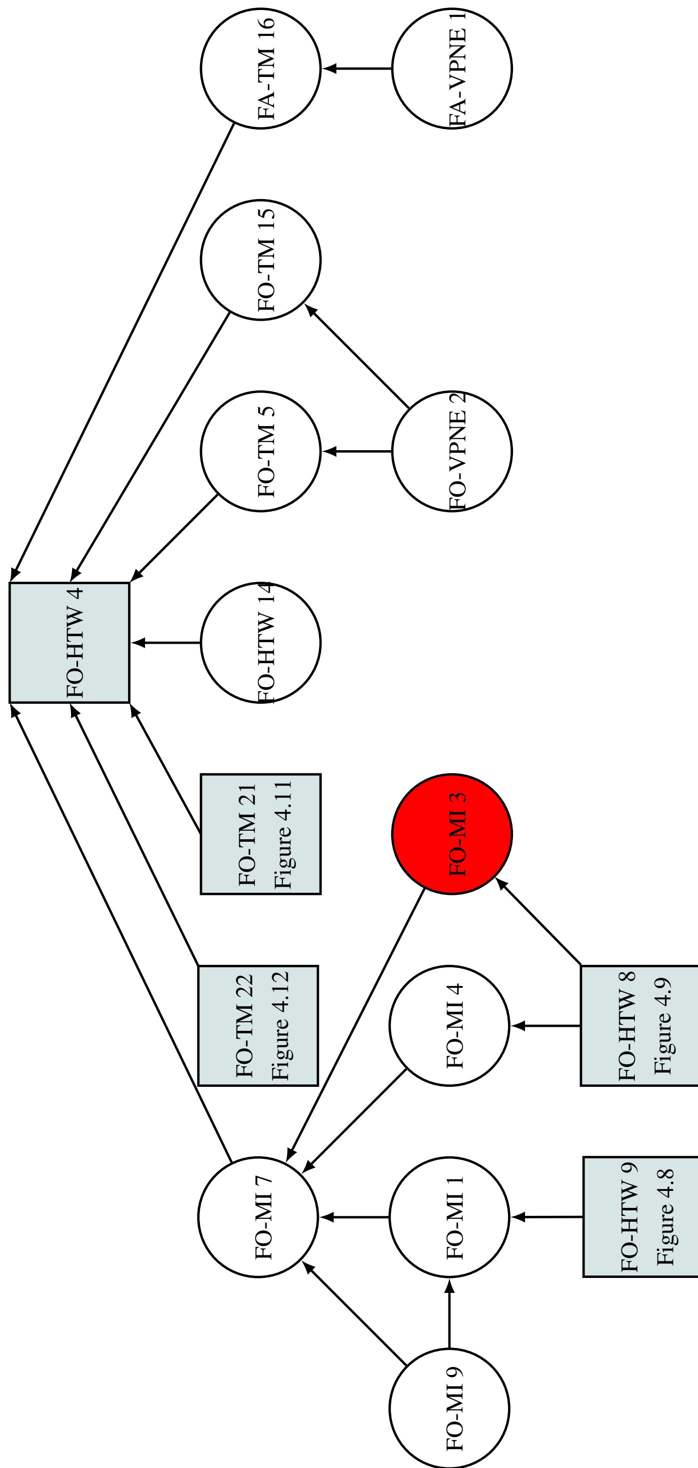

Figure 4.7: Failure effects that lead to the output of corrupted data by the HTW block.

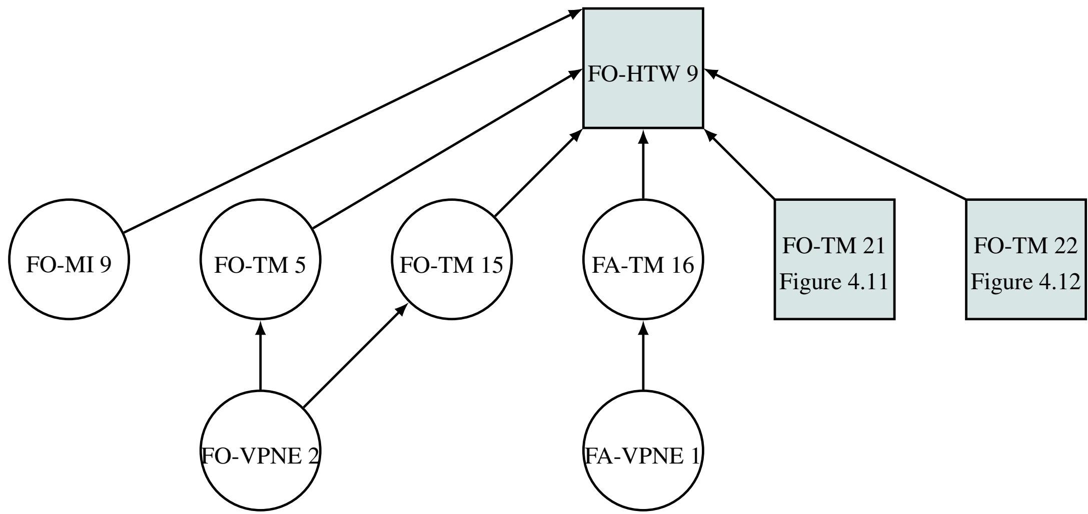

Figure 4.8: Failure effects that lead to the writing of corrupted data to main memory by the HTW block.

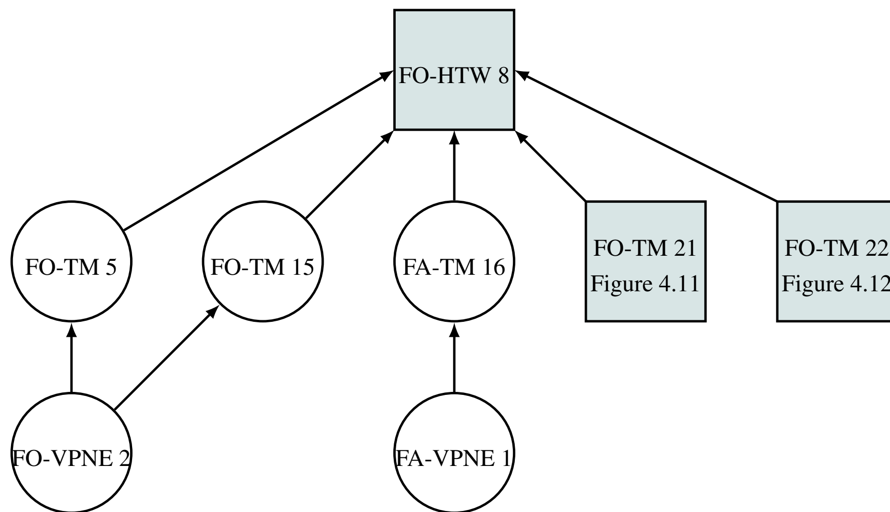

Figure 4.9: Failure effects that lead to the access of a wrong memory address by the HTW block.

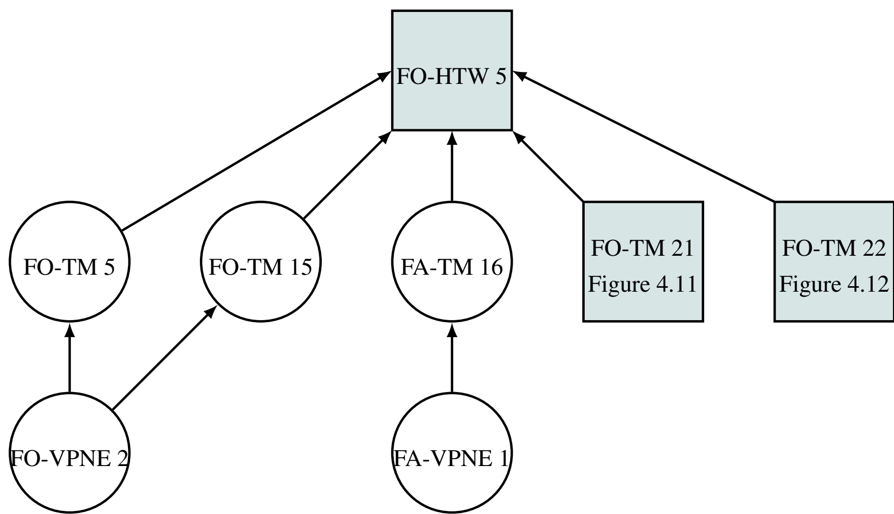

Figure 4.10: Failure effects that lead to the writing of a wrong but valid entry back to memory by the HTW block.

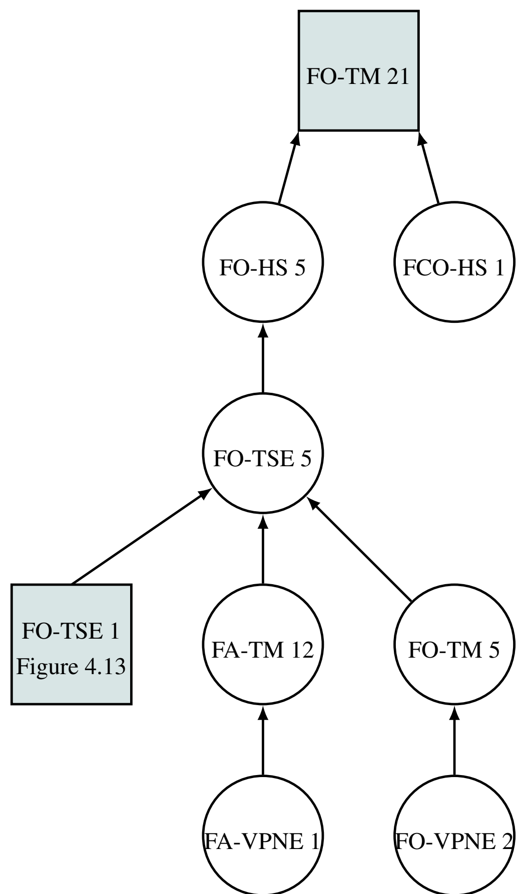

Figure 4.11: Failure effects that result in a corrupted synchronize request by the TM block.

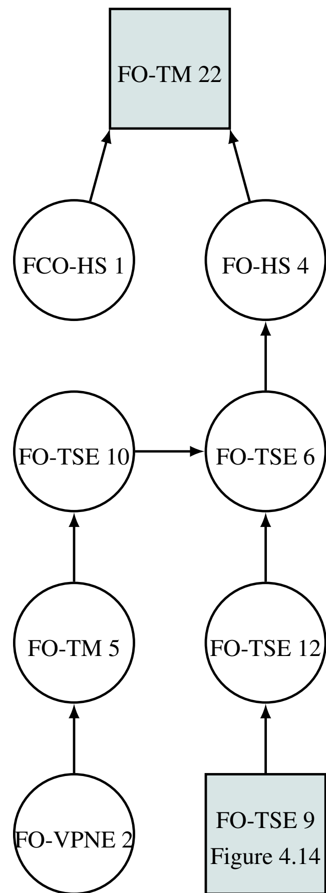

Figure 4.12: Failure effects that result in a synchronize request with a wrong but valid entry by the TM block.

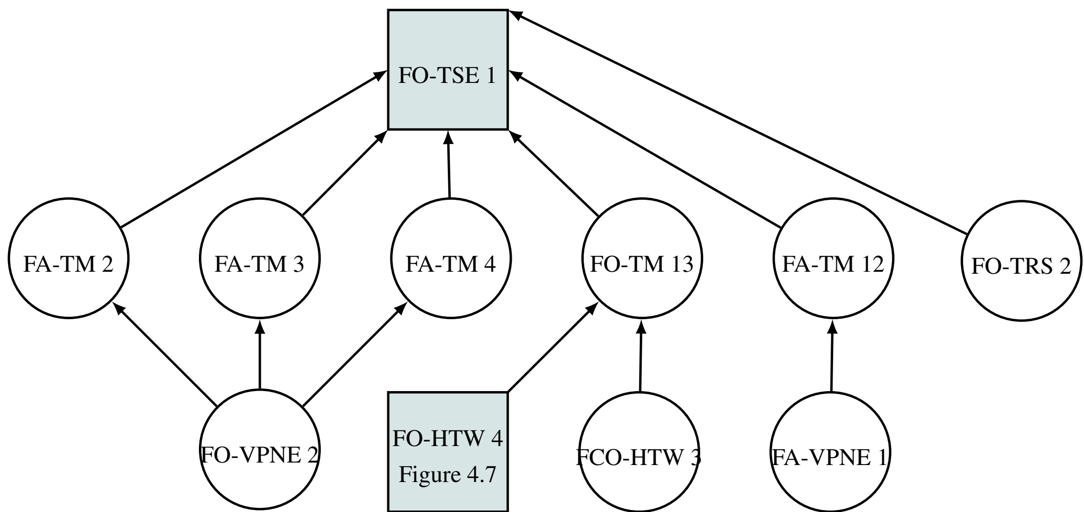

Figure 4.13: Failure effects that lead to corruption of the stored data at a TSE block.

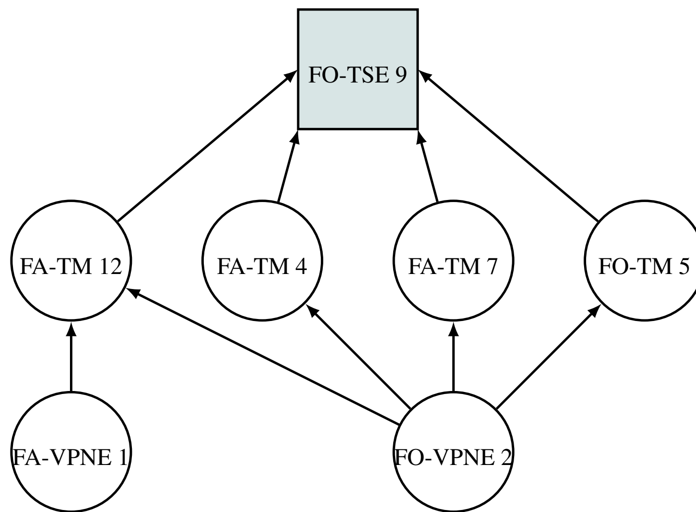

Figure 4.14: Failure effects that lead to the result that a entry stays valid at a TSE block although it should be flushed.

4.5 Conclusion of the Failure Mode Analysis

There are many possibilities for MMU failures with possibly severe consequences, like accessing a wrong memory address. Thus, a MMU without error detection may fail in an unsafe way and therefore is, depending on the actual failure rates, inapplicable for safety-critical systems. Nevertheless, there are many potential error detection and even error correction possibilities that may provide sufficient error-detection coverage to allow the usage of MMUs in safety-critical environments (see Chapter 6).

CHAPTER 5

Fault Injection Experiments

The failure modes of the hypothetical MMU in chapter 4 represent the result of an in-depth failure mode analysis of a theoretical MMU. In order to verify these theoretical results a fault injection in a real MMU is performed. It allows to confirm, extend or refute the theoretical analysis and may also provide statistical data about the relative frequency of certain failure modes.

The following chapter gives a general overview of fault injection possibilities, describes the implemented fault injection in detail and presents the results thereof. Note that this chapter uses the term “context” synonymous to “PID” because the documentation of the LEON3 uses this wording.

5.1 Goals of the Fault Injection

The expected outcome of this fault injection is the evaluation of the theoretical analysis in chapter 4. It shall identify the presented failure modes that are actually seen at the outputs of the MMU and shall reveal any additional failure modes that were not covered in theory. Furthermore, possible improvements for the theoretical analysis shall be detected. However, it is not a target to provide an actual failure rate forecasting for a specific MMU implementation.

The fault injection is completely based on simulated signals, therefore no probabilities for the actual rate of SETs are used. In a hardware-implemented MMU this rate depends on various parameters, e.g., the electrical signal length, the driver strength and various technology parameters. As the goal of this work is to show the principle ways in which a MMU may fail, the focus is not on absolute failure rate prediction of a specific hardware implementation. If it is desired to get the failure rate prediction of such an implementation, the signal list has to be weighted by probability-values. This comes with a considerable effort and should thus be realized with tool support.

Furthermore, the fault injection is limited to signals within the tested MMU. This means that the input signals from the test bench to the MMU are not forced. This is because the analysis

was restricted to the MMU behavior. In real world scenarios faulty inputs from the processor would play a role and have to be considered, but this is out of the scope of this thesis.

5.2 Method

As already described in section 1.3.1, a fault may lead to an error which in turn, if activated, leads to a failure of the system. Thus, fault injection experiments are a possibility to evaluate the robustness of a certain system by controlled injection of faults in the system, while monitoring the system's behavior. This allows us to learn about the way a system may fail under the influence of faults like, e.g., hardware defects or SEE.

According to [GMB+99] the experiment-techniques can be categorized in the following three main groups:

- Software implemented fault injection: The injection is carried out at the information level in order to produce errors in the software as they would be caused by faults in the hardware.
- Physical fault injection: This technique provokes faults in hardware by direct influence on it. This type can be grouped in two subtypes
  - Physical fault injection with contact: Direct influence on the system by, e.g., producing voltage or current changes at the pins of the target chip.
  - Physical fault injection without contact: Indirect influence on the system by some physical phenomenon like, e.g., heavy-ion radiation or electromagnetic interference.
- Simulated fault injection: A model of the target system is simulated and the faults are injected by altering the state of the system. For example, the VHDL model of a given system is simulated, while the logical values of its signal lines are modified in the same way as it is expected to happen due to SEE.

As explained in the next section, this thesis uses simulated fault injection to evaluate the theoretical failure mode analysis of chapter 4 by experimental results in section 5.6.

5.3 Considerations and Design Decisions

The simulation based approach was chosen due to its more reasonable effort compared to physical fault injection for the given subject. Furthermore, the analysis of the hypothetical MMU was performed with a single-fault assumption w.r.t. faults within the MMU. This single-fault assumption may be violated by physical fault injection without contact, because it is possible that multiple signals are influenced at a time by the fault injection. This leads to the possibility that the results may differ from the theoretical analysis. Physical fault injection with contact at a Field Programmable Gate Array (FPGA) device is limited to input faults and thus is not suitable for analysis of internally caused misbehavior. Another possibility to implement a physical fault injection would be to augment the tested system with saboteurs or mutants in order

to alter the internal state. Nevertheless, this type of fault injection targets to test specific functionalities within the system and full coverage on all signal paths is hard to achieve. A software implemented fault injection w.r.t. MMUs is also limited to input faults and hence not suitable.

Due to the fact that the theoretical analysis focuses on transient faults, like, e.g., SEEs, and those are expected to occur far more often in real systems, our fault injection uses this type of faults.

The chosen target for the injection experiments is the MMU of the Aeroflex Gaisler LEON3 processor. This processor was selected due to its free availability in VHDL source code and its usage in space applications. The space industry is known for its conservative approach on performance as a trade-off for high reliability. Thus, the MMU is expected to be a straightforward implementation without complex performance enhancing features, that would make the analysis of the injection results even more complicated. This MMU provides the features presented in section 3.4 and uses a physical and virtual address space of 32 bit.

Due to the availability of this MMU as a VHDL source code, the simulated fault injection is possible on two levels of abstraction. First of all, it can be performed at RTL without further timing information. Second, it can be performed on a synthesized netlist with accurate timing information for a specific target technology like, e.g., a specific FPGA device.

The advantage of the first possibility is that it is faster and the experienced failure modes can be mapped to their causes more easily, because it is a direct simulation of the VHDL source code. Nevertheless, the disadvantage is that it is less realistic and may not reveal some failure modes that can only be observed by the netlist simulation, like, e.g., different input perception due to signal propagation delays. This simulation in turn needs the fixation of a specific target technology, which comes with the price that certain failure modes may be dependent on the target technology.

Additionally, the synthesis process makes it harder to understand how a specific fault causes an experienced failure mode. Nevertheless, the netlist simulation allows to consider additional failure modes like, e.g., different input perception at multiple blocks, which was also covered by the theoretical analysis. Thus, in order to evaluate the theoretical failure mode analysis, fault injections during netlist simulations were chosen for the task at hand.

The fault injection is performed by using the Mentor Modelsim force command to flip and freeze the value of a single signal inside the tested MMU for a specific time during a simulation run. In order to do so, the value of the signal is first read and stored. Then the original driver is disconnected and the signal value forced to the opposite of the read one. After the injection duration is over, the signal is reconnected to its original driver. This type of fault injection is termed signal manipulation and described in [JAR+94]. It corresponds to SETs in hardware. Subsection 5.5.2 presents more details on the work flow of the fault injection.

Another possibility would be to directly flip the values of storage elements, which corresponds to SEUs in hardware. However, due to the fact that flipped signal values are expected to be latched by the storage elements, the signal manipulation also forces this type of errors. The direct storage manipulation alone is not sufficient, because this type is not able to cause failures of combinational elements. Nevertheless, one has to mind that flipped signal values only cause direct effects in the storage elements if the injected pulse lasts until it is latched by the storage element. Thus, faults are partly masked out by this effect termed latch window masking.

As already mentioned, the simulation is a post-layout simulation of the MMU. The netlist was synthesized for an Altera Corporation Cyclone IV FPGA, with part number EP4CE30F29C6, by the synthesis tool Altera Quartus. The MMU inputs and outputs are constrained with board delays of zero time, because they are directly connected to the test bench after the synthesis. Nevertheless, all inputs and outputs are mapped to pins of the FPGA device and hence these signal delays are taken into account. One has to keep in mind that in a real CPU the MMU would be directly connected to the processing unit or the cache system and the memory interface via the FPGA interconnect and thus may allow a higher frequency.

The synthesis tool reports a maximum frequency of 90 MHz for the given setup. In order to maximize the effects of the fault injection, this frequency is chosen as the clock frequency for the test bench and the MMU.

Furthermore, the MMU is configured to use two fully-associative TLB entries which can both be used for data and instruction TLB entries. This is the smallest possible configuration of this MMU w.r.t. the number of internal signals. The implemented fault injection targets to influence every internal signal of the MMU at least once in order to cause all possible failure effects. Thus, the necessary number of fault injections and hence the duration of the fault injection experiments depends directly on this number. Nevertheless, it is not expected that a larger TLB causes additional failure modes. This allows the fault injection to keep full coverage of the failure modes while reducing the duration of the experiments, which is relevant because it is in the order of days. The decision against separated TLBs for data and instruction translations was determined by the fact that the hypothetical MMU, considered in chapter 4, does not differentiate between the two types. In order to cover the replacement strategy discussed in the theoretical analysis, the replacement strategy for TLB entries is configured to be LRU.

Another possibility to keep the duration of the fault injection experiments as short as possible is to cut down the workload that is used for the simulation. Nevertheless, the minimal workload is determined by the number of features that need to be tested and hence cannot be reduced arbitrarily.

5.4 Tools

The number of signals within the netlist of the LEON3 MMU is in the order of ten-thousands and therefore a manual fault injection is impractical. Thus, a number of tools were developed in order to ease and automate this task. As already mentioned, there is a test bench needed to interface the MMU. In addition, a Graphical User Interface (GUI) was implemented to perform the actual fault injection. Furthermore, another GUI allows to check the large number of translation results for failures due to the fault injection.

The basic structure of the fault injection setup is shown in figure 5.1. Detection of failures due to the fault injection is performed by comparison of the results of two identical MMUs, the tested MMU and the reference MMU, which is also termed golden MMU. In order to eliminate common cause failures, each of those MMUs has its own stimulus generator (MMU Stimulus) attached. This is needed due to the fact that the stimulus generator has to respond to the output signals of the MMUs, but the tested and the golden MMU may produce different output signals due to the fault injection. For example, it is possible that the golden MMU indicates a read

access to the memory whereas the tested MMU indicates a write access due to a flipped internal signal.

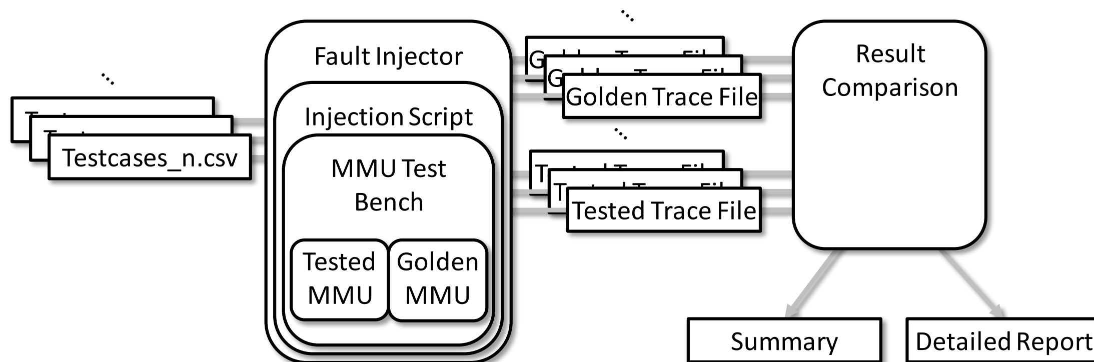

Figure 5.1: Structure of the fault injection setup

5.4.1 LEON3 MMU Test Bench

The fault injection is confined to the MMU. Thus, it is legitimate to directly control the MMU via a VHDL test bench and to consider the input signal vectors to the MMU, which are directly driven by the test bench, as fault free. Hence, the inputs are neither recorded nor further evaluated during and after the fault injection. Nevertheless, the direct inputs to the MMU are transmitted via signals inside the MMU and hence it is possible that inputs are altered inside the MMU by the fault injection. Note that the fault injection may cause values unequal to logical one or zero at signals of the nine-value types std_ulogic, std_logic and its corresponding vectors. This values are termed meta-values within this thesis.

All output signals of the MMU are recorded on the rising clock edge, whenever a result is indicated, a memory request is indicated or the indicator signal shows a meta-value, and evaluated later on by a separate tool (see 5.4.3). Furthermore, also the memory input signal to the MMU is recorded whenever a response is indicated on the rising clock edge.

In addition to the pure recording of the signal values in the trace files, the test bench categorizes different output states on the rising clock edge and stores corresponding code values in the trace file. The following list contains the code values on the left side:

D1 The signal *mmudc_out.transdata.finish* is active and *mmudc_out.transdata.accexc* is inactive. This means that a data translation result with no access exception is flagged.

D2 The signal *mmudc_out.transdata.finish* as well as *mmudc_out.transdata.accexc* are active. This means that a data translation result with an access exception is flagged.

I1 The signal *mmuic_out.transdata.finish* is active and *mmuic_out.transdata.accexc* is inactive. This means an instruction translation result with no access exception is flagged.

I2 The signal *mmuic_out.transdata.finish* as well as *mmuic_out.transdata.accexc* are active. This means an instruction translation result with an access exception is flagged.

FS The signal *mmudc_out.mmctrl2.valid* is active. This means a new fault in the fault status register is flagged. Note that this applies to data as well as instruction translations.

MR The signal *mcmm_in.req* as well as *mcmm_in.read* are active. This means a table walk read operation is requested.

MW The signal *mcmm_in.req* is active and mcmm_in.read is inactive. This means a table walk write operation is requested.

M1 The table walk request signal *mcmm_in.req* contains a meta-value.

M2 The data translation result indication signal *mmudc_out.transdata.finish* contains a meta-value.

M3 The instruction translation result indication signal *mmuic_out.transdata.finish* contains a meta-value.

M4 The signal *mmudc_out.mmctrl2.valid* that indicates a new fault in the fault status register contains a meta-value.

T A translation timeout occurred.

R The signal *mcmm_out.ready* that indicates a table walk response is active.

M5 The signal *mcmm_out.ready* that indicates a table walk response contains a meta-value. Note that this signal is controlled by the memory interface of the test bench and thus should never contain a meta-value.

M6 The signal *mcmm_in.req* is active and the signal mcmm_in.read that determines a read or write table walk request contains a meta-value.

M7 The signal *mmudc_out.transdata.finish* is active and the signal *mmudc_out.transdata.accexc* that indicates a data translation access exception contains a meta-value.

M8 The signal *mmuic_out.transdata.finish* is active and the signal *mmudc_out.transdata.accexc* that indicates an instruction translation access exception contains a meta-value.

M9 The signal *mmuic_out.grant* that indicates that a instruction translation was granted contains a meta-value. This signal is set by the MMU to acknowledge an instruction translation request. Note that this signal is only recorded in case of an meta-value. In case the signal indicates that the request was granted, the test bench responds by setting the request indicator to inactive and waits for the translation result. If the signal never indicates that the request was granted, the translation eventually runs in a timeout.

M10 The signal *mmudc_out.grant* that indicates that a data translation was granted contains a meta-value. This signal is set by the MMU to acknowledge a data translation request. The same limitations as for the *mmuic_out.grant* signal, mentioned in M9 apply to this signal.

The test bench implementation is based on [Eis02], analysis of the VHDL code and Mentor Modelsim simulations. The base structure is shown in figure 5.2. The signal vectors *mmudc_in* and *mmuic_in* are inputs to the MMU and are used to perform a request for either a data address translation or an instruction address translation but are not recorded in the trace file. The corresponding recorded results are output at the *mmudc_out* and *mmuic_out* signal vectors. Furthermore, the signal *mcmm_in* is used as input to the emulated memory that handles memory write and read requests and thus equates to an output of the MMU. Hence, this signal vector is also recorded by the test bench. The corresponding output of the emulated memory is the *mcmm_out* signal vector, which is used to provide the MMU with the requested table walk data. The emulated memory only responds to valid memory addresses with valid PTDs and

PTEs. Otherwise an invalid marked entry is returned. Faults at this interface are unveiled by the recording of the *mcmm_in* signal vector, the *mcmm_out* signal is driven by the test bench but nevertheless recorded to ensure that no difference occurs without a preceding difference at the *mcmm_in* signal. Nevertheless, during the fault injection, two such differences occurred. This was caused by the effect described in section 5.6.5.2 and thus is not considered a problem in the test bench.

The test bench inputs are stored in a Comma-Separated Values (CSV) file for comfortable input manipulation. This CSV file also contains corresponding expected response values like, e.g., the translation results and the fault status of the MMU. The expected responses were used during the implementation of the test bench in order to check if the test bench interfaces the MMU in a correct way and if the behavior of the MMU is correctly understood.

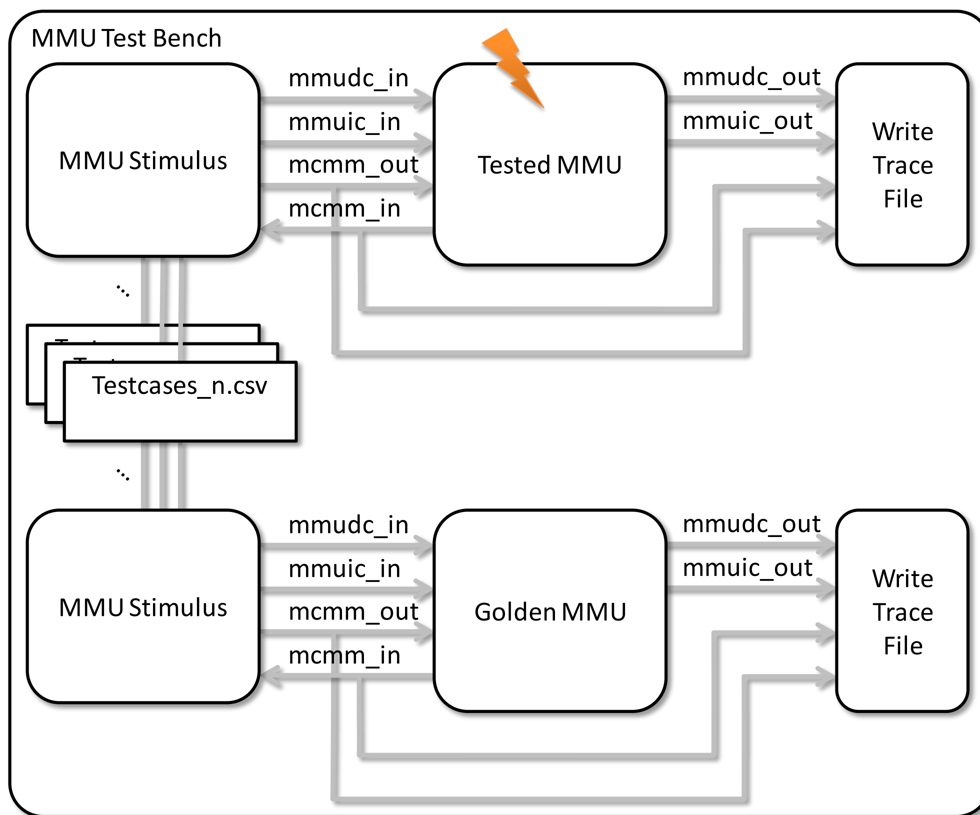

Figure 5.2: Structure of the VHDL test bench

5.4.1.1 Deviation of the LEON3 MMU from its Data Sheet

During implementation of the test bench, it was experienced that the LEON3 MMU actually only uses 32 bit physical addresses instead of 36 bit as stated in the data sheet. This behavior was

confirmed by Aeroflex Gaisler. The regarding e-mail correspondence is quoted in appendix A.

5.4.2 Fault Injection GUI

A GUI was developed using Qt/C++ to perform the fault injection experiments. It allows comfortable changing of various relevant parameters and interacts with Mentor Modelsim via Tool Command Language (Tcl) scripts to perform the simulation runs.

The GUI allows to compile the test bench together with the synthesized MMU design and extract the signal list (see figure 5.3), which in turn is used for the fault injection. Furthermore, various fault injection settings can be set (see figure 5.4). And finally the fault injection can be started and some status information is provided (see figure 5.5).

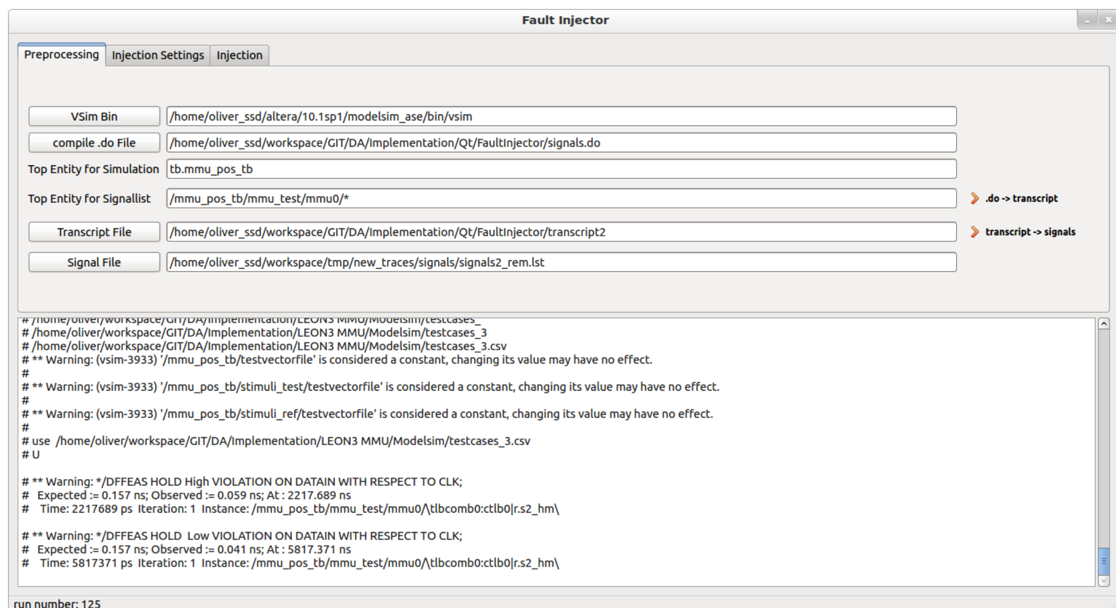

Figure 5.3: Fault Injection GUI: Tab for compilation of the MMU design and extraction of the signal list.

5.4.3 Result Comparison GUI

Another GUI tool was developed to evaluate the large amount of trace files resulting from the fault injection experiments. This tool is also written in Qt/C++ and is shown in figure 5.11.

As mentioned in section 5.4.1, all output signals of the MMU are recorded in these trace files on the rising clock edge, whenever a result is indicated, a memory request is indicated or the indicator signal shows a meta-value. Furthermore, also the memory input signal to the MMU is recorded whenever a response is indicated on the rising clock edge and all signals are directly evaluated by the test bench in order to additionally store corresponding action codes in the trace files.

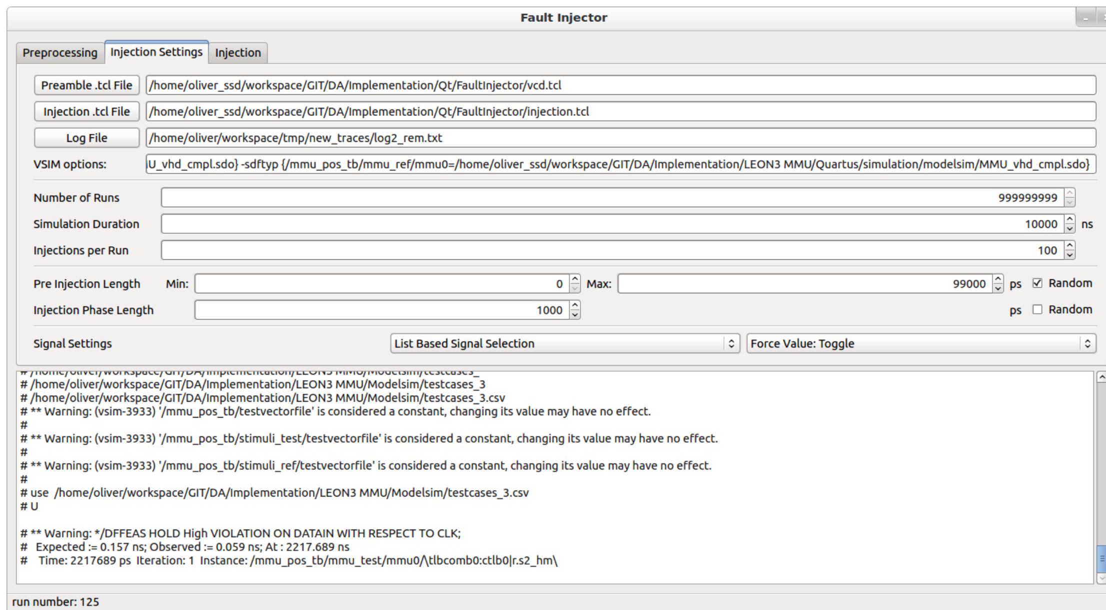

Figure 5.4: Fault Injection GUI: Tab for the fault injection settings.

Figure 5.6 shows an example trace that is part of a trace file. The contents are accordingly marked in the figure. Note that the workload and workload block numbers are numbered beginning with zero in the trace file. Thus, the example trace shows part of workload block 1 in workload 1. The details about the workloads and workload blocks are presented in section 5.5.1.

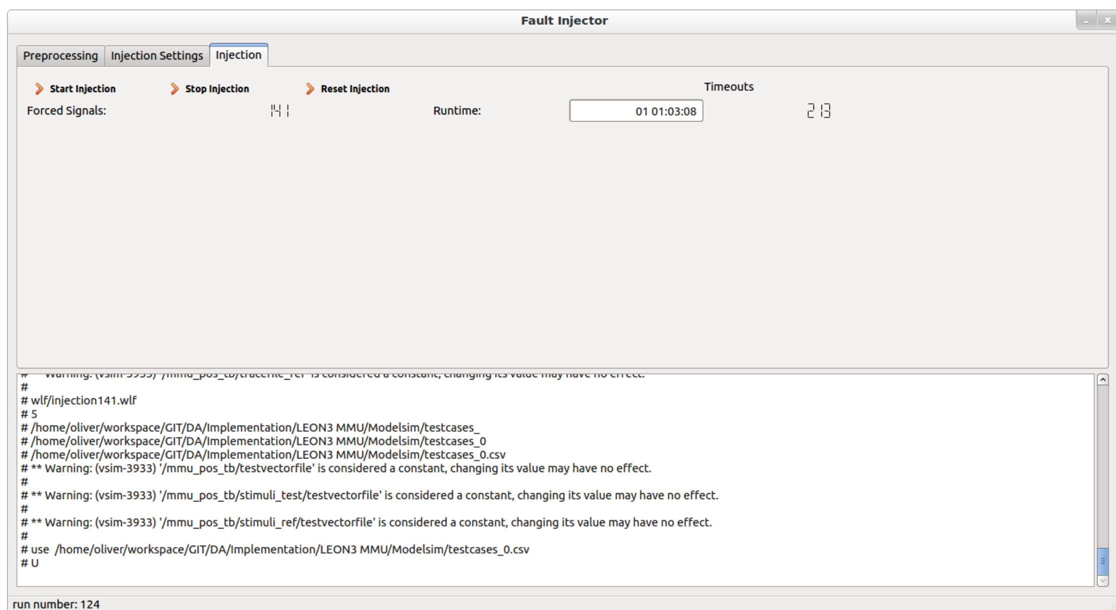

Figure 5.5: Fault Injection GUI: Tab for the actual fault injection.

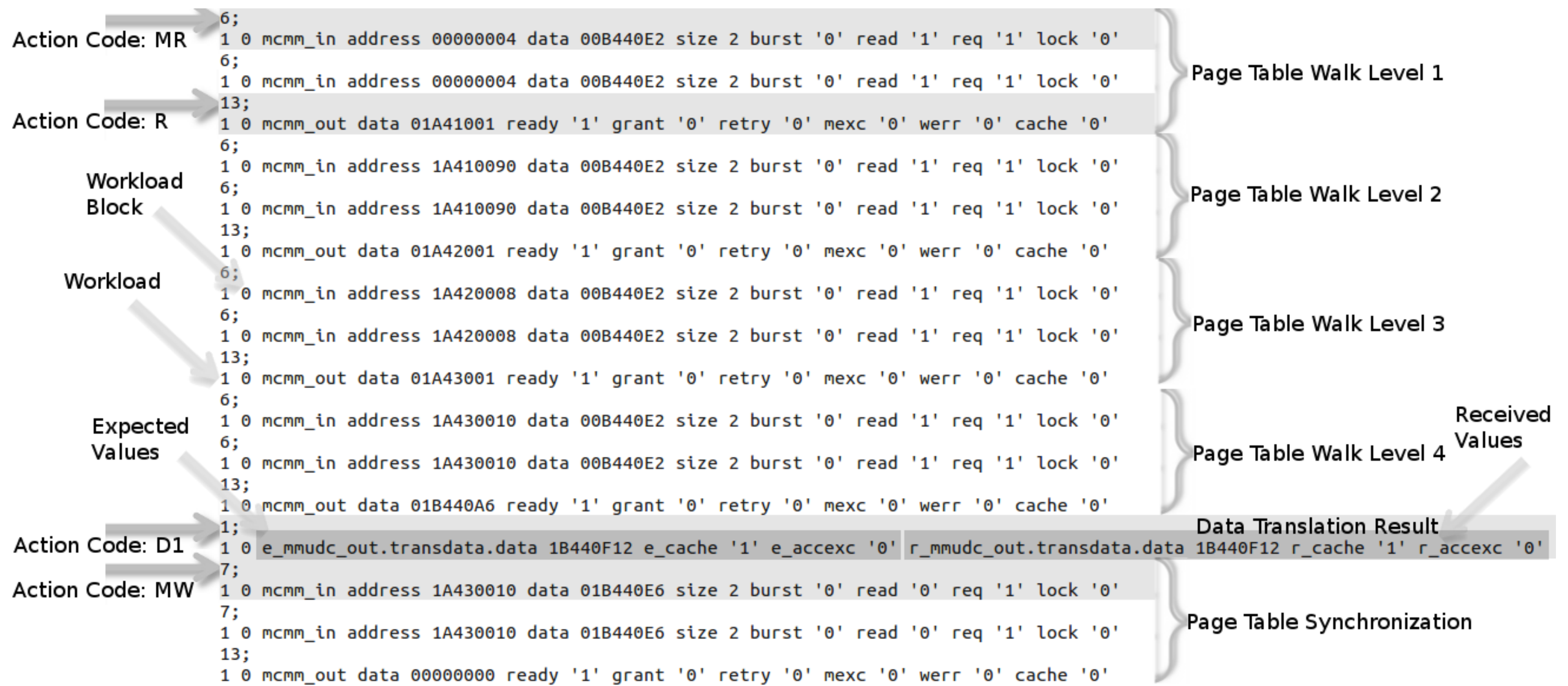

Figure 5.6: Example trace showing a data translation with a TLB miss, the corresponding page table walk, the data translation result and a page table synchronization afterwards

The result comparison GUI generates two files, one summary file that contains various statistics of all used workloads, used workload blocks and failure types and one file that contains the number of each failing run, the forced signal and which output signal deviated.

The comparison is line and character based. The trace files contain lines for the performed action code and lines for the signal values alternating. At first, the line for the action code of the reference MMU is compared to the action code of the tested MMU. In case a difference is detected, the action code at the tested MMU is correlated to the single action code of the reference MMU and each action code of the tested MMU is reported as a first detected deviation. If the reference action code is contained in the action code of the tested MMU the signal value line is considered comparable. Figure 5.9 shows a trace where the action code differs. This trace increments the count in column MR of row MW in table 5.9 by one and the signal value line is considered not comparable.

In case the signal value line is considered not comparable, the following signal value line is only checked for signal values of 'X' or signal vectors containing 'X'. Each of them is counted once. As explained in section 5.6.4.1, values of 'X' are a potential risk and thus counted separately. The evaluation of the deviating trace file is stopped after that. Figure 5.7 shows a trace where the tested MMU requests a table walk operation with a value of 'X' in the read or write signal. The signal value line is considered not comparable due to the differing action codes and contains a value of 'X'. Hence, this trace leads to an increment of the count of 'X' by one and increments the count in column M6 of row MR in table 5.9.

Figure 5.8 shows a trace where the tested MMU requests a table walk read operation with a value of 'X' in the address signal. The signal value line is considered comparable and contains values of 'X'. Hence, this trace leads to an increment of the count of 'X' by one, although actually two 'X' are contained in the address signal value and also increments the number of detected first differences for the memory read request address. Afterwards the comparison is stopped.

```
3 6;                         Golden MMU
4 0 0 mcmm_in address 00000000 data 00000020 size 2 burst '0' read '1' req '1' lock '0'
```

```
← 3 15;                       Tested MMU
  4 0 0 mcmm_in address 00000000 data 00000020 size 2 burst '0' read 'X' req '1' lock '0'
```

Figure 5.7: Example trace showing a deviating action code and 'X' in the trace of the tested MMU

```
73 6;                         Golden MMU
74 4 2 mcmm_in address 5A42000B data 00000020 size 2 burst '0' read '1' req '1' lock '0' →
```

```
← 73 6;                       Tested MMU
  74 4 2 mcmm_in address XAX20000 data 00000020 size 2 burst '0' read '1' req '1' lock '0'
```

Figure 5.8: Example trace showing 'X' in the trace of the tested MMU

```
105 7;                        Golden MMU
106 3 3 mcmm_in address 6A430010 data 06B440FA size 2 burst '0' read '0' req '1' lock '0'
```

```
← 105 6;                      Tested MMU
  106 3 3 mcmm_in address 00000004 data 06B440BA size 2 burst '0' read '1' req '1' lock '0'
```

Figure 5.9: Example trace showing a deviating action code in the trace of the tested MMU

Action Code M2
Action Code MR
Redundant Signal Value Information
Comparable Signal Values

```
79 6;                         Golden MMU
80 3 3 mcmm_in address 6A420008 data 05B440F6 size 2 burst '0' read '1' req '1' lock '0'
```

```
← 79 9;6;                     Tested MMU
  80 3 3 d m3 3 mcmm_in address XAX10090 data 05B440F6 size 2 burst '0' read '1' req '1' lock '0'
```

Figure 5.10: Example trace showing a deviating action code in the trace of the tested MMU together with the expected action code and 'X' in the address signal

After the successful comparison of the action code line, i.e., the action code of the reference MMU is contained in the trace of the tested MMU, the signal value line of the tested MMU is compared to the signal value line of the reference MMU. Due to the comparison of the action codes before, those lines are always comparable, but it is possible that the tested MMU line contains more values than the reference line. In this case, the tested MMU line is truncated to the comparable parts. The values in the line are then compared and each differing value is reported as a data signal deviation. Furthermore, the check for 'X' is performed as described before. Note that signal vectors are represented as hexadecimal values and only one deviation is reported, no matter how many of its signals actually deviated. Once a deviation in a signal value line is detected the comparison of the trace file is stopped. Figure 5.10 shows a trace where the action code of the reference acMMU is contained in the action code of the tested MMU but an additional action code (M2) is reported. The signal value string is trimmed to the comparable part and a difference in the address is reported. Furthermore, the count of 'X' is incremented by one.

This algorithm is performed for every line of every trace file. Nevertheless, the tool cannot evaluate more than one differing action code and signal value line per trace file. First of all, there is no possibility to detect which difference is actually caused by the fault injection and which one is a consequence of a previous difference. Second, if, e.g., the compared trace files are shifted due to the fault injection, all following lines would be counted as faults. This drawback could possibly be minimized by a more elaborated comparison algorithm, but this would go beyond the scope of this thesis.

The advantage of the trace file comparison in contrast to a direct result evaluation by the test bench is, that it eliminates the need for absolute clock synchronous operation of both MMUs. This means that the results are considered as equivalent even if they appear in a different clock cycle, as long as the result order is the same.

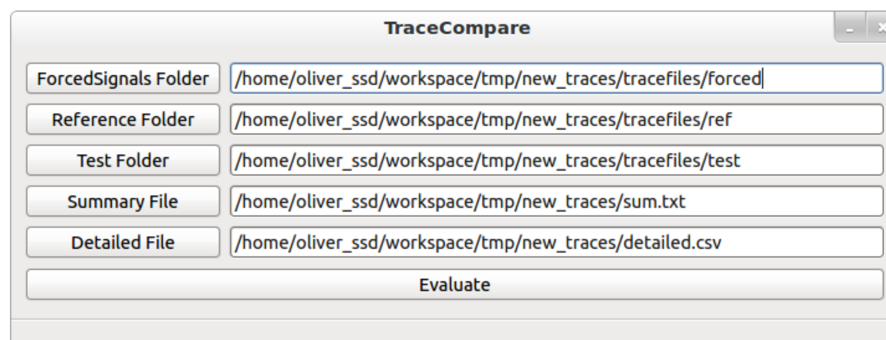

Figure 5.11: Result Comparison GUI

5.5 Parameters

In order to allow evaluation of the results generated by the MMU during fault injection, a proper workload has to be provided and the fault injection has to be configured. These steps are crucial for the fault injection and are thus presented in detail in the following section.

As the instants of simulation time when the faults are injected, are selected at random, a longer workload together with more fault injections would have a higher probability to observe faults and thus a bigger coverage of all possible failures. Nevertheless, the simulation duration is directly proportional to these parameters. As a tradeoff, the injection instants are chosen at random, but, in order to distribute the injections over the whole workload run more evenly, the simulation is split in predefined intervals. During each of those intervals only one injection is performed at a randomly selected point in time.

The workload covers all relevant functions of the MMU at least once, but the different functionalities vary in the number of needed clock cycles until the result is ready. Hence, one has to keep in mind that more complex functions and functions that are used more frequently by the workload, have a higher possibility to be influenced by the fault injection.

5.5.1 Workload of the Test Bench

The workload of the test bench is divided in multiple blocks, each of them targeting to test specific functionalities of the MMU. The enumeration below gives an explanation of the workload blocks.

1. Access Permission Test (0 - 4): This block tests the functionality of the access permission checking by the MMU. All of the following translation requests are performed for TLB entries with access permission fields from zero to four. Table 5.1 lists the corresponding access permissions for each of the access permission field values.

    - User read access
    - User write access
    - User execute access
    - Supervisor read access
    - Supervisor write access
    - Supervisor execute access

    In detail this test performs the following actions:

    a) Issue the first translation request for a unique context. This invokes a hardware table walk for the corresponding PTE.

    b) Supply a PTE with the intended access permissions and attributes via the memory interface.

    c) Issue all remaining different translation requests within the previously used page, but with different virtual addresses.

	Accesses allowed	
Access Permission Field	User Access	Supervisor Access
0	Read Only	Read Only
1	Read/Write	Read/Write
2	Read/Execute	Read/Execute
3	Read/Write/Execute	Read/Write/Execute
4	Execute Only	Execute Only
5	Read Only	Read/Write
6	No Access	Read/Execute
7	No Access	Read/Write/Execute

Table 5.1: Access permission fields for user and supervisor software at page table entries of Aeroflex Gaisler LEON 3, based on [SPA92, p. 243].

d) Read the fault status in case there is a fault expected. This clears the fault status register for the next translation. Note that in case there is an unexpected fault reported due to the fault injection, this is recognized by the evaluation of the trace file. Hence, there is no need to read the fault status register after every translation request, which saves simulation time.

e) Repeat these steps for the access permission field values from zero to four. In order to cause a page table walk for all different access permissions, each of the permission tests uses a unique context.

Note that in step 1b also the different attribute bits of the PTEs are tested. Each of the unique contexts results in another attribute bit configuration for the cacheability, modified and referenced flags. Each of those configuration bits is covered once and another test is performed where all bits are inactive.

Due to the fact that the MMU is configured to use two combined TLB entries, three entries have to be replaced during this test block. As verified by the simulation, write requests to TLB entries that are marked as not modified and requests to not referenced PTEs automatically cause an update of the corresponding TLB entry in memory. Thus the writing and reading parts of the hardware table walk are tested by these cases. Furthermore, if it is expected that an access violation is flagged, the flag is read and thus cleared by the workload after the translation request. Hence, also this functionality is tested.

2. Flush Function Test: This block tests the flush function at all four different page table levels, page, segment, region and context together with a flush of the entire TLB (see [Eis02, p. 31] for details). The test is performed by a translation request, followed by a flush request for one of the levels and afterwards by performing another translation request for the same virtual address and context but with the hardware table walk provided with a different page frame address. Thus, due to the necessity of a table walk, the translation has to result in a different physical address than before the flush. Hence, a malfunction of the

flush function can be detected if the golden and tested MMU provide different translation results.

3. Memory Exception Test: During this test block the test bench signals a memory exception once at every level of the page table to the hardware table walk. Thus, the MMU is expected to raise a corresponding error flag, which in turn is cleared by the workload and followed by a TLB flush to gain a fresh starting point for the next memory exception test. This test is repeated for all four levels.

4. Access Permission Test (5 - 7): This is the same test workload as in point 1, besides that it uses access permission field values from five to seven.

5. Different Translation Levels Test: This block performs all the translation requests listed in point 1 but with the hardware table walk configured to return either a level 1 (context), a level 2 (region) or a level 3 (segment) PTE. In order to cut down the simulation time, it only uses one access permission field value, from five to seven, for each of the levels. The level 4 (page) test can be skipped because this is already covered by point 1.

6. Page Fault Test: This block tests the page fault behavior of the MMU. It performs translation requests with the table walk provided with an invalid marked PTD/PTE once for each level of the page table.

The order in which the workload blocks are executed may influence the recognized failure modes and thus can represent a systematical problem in the fault injection. To eliminate this possibility, six different workloads are used. Each of them contains the aforementioned workload blocks in a shifted arrangement. E.g. the first workload starts with the access permission test (0-4), the second workload starts with the Flush Function Test and contains the Access Permission Test (0-4) at its end. The different workloads are chosen at random by the Tcl script that performs the fault injection.

Table 5.2 maps the workload blocks to the LEON3 MMU functionalities. Furthermore, it states if the corresponding functionality was also covered by the analysis of the theoretical MMU in chapter 4. Due to the fact that the theoretical MMU is of lower complexity than the LEON3 MMU, some more advanced functions are not covered there. To be more specific, the theoretical MMU provides no possibility for multi-level page tables, non-executable memory addresses and memory mapped IO.

feature name	workload block	existing at hypothetical MMU
data address translation	1, 2, 4, 5	yes
data address translation at different levels	5	no
instruction address translation	1, 4, 5	no
instruction address translation at different levels	5	no
different access permissions	1, 4, 5	yes
different contexts	1, 4, 5	yes
page cacheable property	1, 4, 5	no
page referenced property	1	yes
page modified property	1	yes
flush TLB entry	2	yes
flush TLB entry at different levels	2	yes
probe TLB entry	none[a]	no
TLB entry replacement	1, 4, 5	yes
successful hardware table walk	1, 2, 4, 5	yes
successful hardware table walk at different levels	5	no
hardware table walk with page fault	6	yes
hardware table walk with page fault at different levels	6	no
hardware table walk with memory exception	3	yes
hardware table walk with memory exception at different levels	3	no
fault reporting	1, 3, 4, 5, 6	yes

[a] as stated in [Eis02], the probe operation is rarely used and thus no effort was taken to cover it.

Table 5.2: MMU functions covered by the workload compared to the coverage of the theoretical analysis

5.5.2 Fault Injection Sequence

In order to ease the description of the injection sequence, some parameters are explained in advance:

- The number of injections per complete run n is 100. This was chosen as a compromise between fault frequency and simulation overhead.
- The simulation time of a complete run for one workload $t_{complete}$ is chosen to be 13.1 μs. It is sufficient to allow a complete run of the longest workload (12.505 μs) in combination with one translation timeout (551 ns). Such a timeout can be caused by the fault injection and every timeout is expected to cause noticeable differences in the trace files. This is because the test bench autonomously resets a MMU that runs in a timeout and issues the next translation request afterwards. Hence, the result of the failed transmission is missing.

  Note that the simulation times of the different workloads vary slightly because some workload block combinations need additional TLB flushes at their boundaries.
- The injection duration t_{inj} is chosen to be 1 ns.

  The focus of this work is on actual failure behavior, therefore the injection duration is not trimmed for maximum coverage of real world scenarios but chosen to be a good compromise between failure rate and realistic SET pulse widths.

  In order to select a satisfactory injection duration, multiple test runs including fault injections on one fixed signal with various injection durations were performed and the failure rates were observed. All other simulation settings were the same as they were used for the results in section 5.6. In order to get more representative results, this test was repeated 20 times for each injection duration. Afterwards the number of trace files with differences to the trace files of the golden MMU was determined for each injection interval. The results are shown in table 5.3. As it can be seen, the failure rate correlates positively with the fault injection duration until the maximum possible is reached.

  Actual SET pulse width measurement results for modern technologies can be found, e.g., in [LKJ$^+$12] and [HMHO12]. In order to keep the duration at a realistic level, the value of 1 ns was chosen for the fault injection simulation that covered all forceable signals of the MMU.
- The pre-injection time t_{pre} is randomly selected between 0 ns and 130 ns. This is caused by the fact that the complete simulation time $t_{complete}$ is separated in $n = 100$ intervals of length 131 ns, during which one injection of duration t_{inj} is performed at a random instant. Hence, there is a pre-injection time t_{pre} before the injection and a remaining time t_{rem} afterwards.
- The relation of the time values is given in equation 5.1.

$$t_{rem} = \frac{t_{complete}}{n} - t_{pre} - t_{inj} \tag{5.1}$$

Injection duration	differences
10 ps	1
100 ps	6
500 ps	9
1 ns	10
5 ns	19
10 ns	20
50 ns	20

Table 5.3: Fault injection experiments on one signal line with various injection durations.

The injection flow used by the fault injection GUI uses the previously described parameters and is implemented as follows:

1. A signal for the fault injection and the workload (see section 5.5.1) is chosen and kept constant for steps 2-8. This is performed on the basis of a list of all signals, which is evaluated line by line, in order to force all internal signals of the tested MMU. Whenever a signal vector is encountered, every signal of the vector is separately used for a complete injection run. The workload is chosen at random for each injection run.
2. The simulation with the workload is run for the pre-injection time t_{pre}.
3. The signal value is read and stored.
4. If a value of '1' was encountered in step 3, the signal is forced to be '0', otherwise a value of '1' is forced. This is only true for signals of type std_logic and std_ulogic. Note that the forced value of '1' also applies to all other possible values of these types. All other signal types are reported as unforceable and are separately included in the statistics.
5. The simulation is run for a specific injection time t_{inj}.
6. The signal value is unforced.
7. The simulation is run for the resulting remaining time t_{rem}.
8. The steps 2-7 are repeated for the number of injections per complete run n.
9. If there are signals left that were not already forced, the steps 1-8 are repeated.

Signals and signal vectors	22382
Forced signals	30993
Unforceable signals and signal vectors	46
Detected first faults at indicator signals	3735 (40.20 %)
Detected first faults at data signals	4915 (52.91 %)
Timeouts	640 (6.89 %)
Sum of detected first faults (including timeouts)	9290
Percentage of detected first faults to number of forced signals	29.97 %

Table 5.4: Fault injection result summary

5.6 Result Evaluation

The fault injection was performed with Mentor Modelsim Altera Starter Edition version 10.0d on a Ubuntu 12.04.1 LTS Linux system with an Intel Core 2 Quad Q6600 processor (four cores with 2.4 GHz), 4 GB of RAM and an Intel SSD320 solid-state drive. Four simulations were run in parallel, leading to a complete runtime of about four days. The resulting data to be analyzed sums up to a total amount of 9.5 GB with 3.3 GB being textual trace files and 6.2 GB of waveform files for in depth analysis.

Table 5.4 shows an overview of the fault injection experiment parameters and results. The first line gives the number of signals and signal vectors of the tested MMU. At this point every signal vector is counted once, independent from its actual length. The separation in multiple forced signals is executed by the fault injection script. The number of forced signals represents the actual count of separated signals that were targeted by fault injections. Some signals, i.e., 0.2 % of all signals and signal vectors, are not of type std_logic, std_ulogic and are considered unforceable by the injection script. Due to the low quantity of signals from other types, no effort was taken to also force those signals. Furthermore, we were quite curious why there are signals of, e.g., type boolean, left after synthesis but did not find an answer to this question.

The number of detected first faults is split into first faults at indicator signals and first faults at data signals, which are going to be evaluated in more detail in section 5.6.5 and 5.6.6. Furthermore, the number of detected timeouts is separately listed and will be described in section 5.6.4.2. For simplicity, the evaluation of the workload and the workload blocks will use the sum over all types.

In order to handle the amount of data generated by the fault injection, tool support was needed, as already described in section 5.4.3. The following section is going to evaluate the results obtained and describe any possible bias introduced by the tools. Furthermore, figure 5.12 visualizes the results obtained.

5.6.1 Influence of the Workloads

In order to make sure that the random selection of the workloads does not introduce a bias, the used workloads are reported in the trace files. Table 5.5 lists the usage count together with

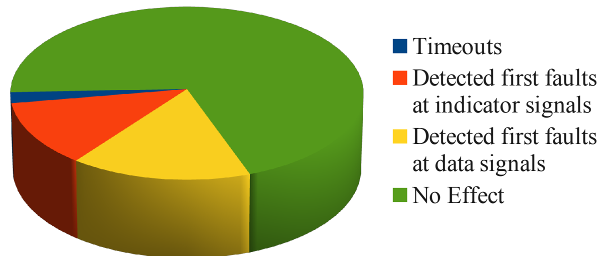

Figure 5.12: Visualization of the fault injection result summary

the percentage w.r.t. all performed injections for each workload. As it can be seen, none was disproportionately often used but workload 2 was slightly more frequently selected.

Parameter	Absolute number	Percentage of all
All workloads used	30993	100.00 %
Workload 1 used	5109	16.48 %
Workload 2 used	5273	17.01 %
Workload 3 used	5190	16.75 %
Workload 4 used	5123	16.53 %
Workload 5 used	5124	16.53 %
Workload 6 used	5174	16.69 %

Table 5.5: Usage summary of the workloads

Figure 5.13 shows the absolute number of detected first faults itemized per workload. The dotted line represents the average number of 1548.3 detected first faults. As it can be seen, no workload caused significantly more or less first faults. The only exception is workload 2, which is due to the fact that it was slightly more often used and thus caused 89.67 detected first faults, corresponding to 5.79 %, more than average, whereas workload 6 caused 2.15 % less detected first faults and represents the minimum. Nevertheless, due to the randomness of the fault injection, those values allow us to conclude that the arrangement of the workload blocks does not directly influence the absolute number of detected first faults.

In contrast, the number of detected first faults per workload block is indeed influenced by the arrangement of the workload blocks within the different workloads. Figure 5.14 shows the number of detected first faults per workload block itemized per workload. It shows a clear spike at one workload for each workload block. Thus, for each workload block there is one workload that clearly causes more detected first faults than the other trace files.

This effect is caused by the comparison algorithm used in the comparison GUI. It only

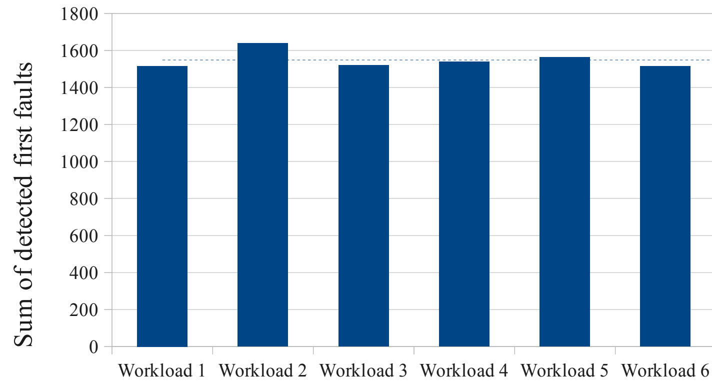

Figure 5.13: Sum of detected first faults w.r.t. workloads

counts the first occurred difference in the trace files. Thus, if there is a difference detected during the first workload block, the remaining blocks do not get evaluated. Due to the shifting of the workload blocks in the workloads, the first workload block is different for each workload. Hence, for every workload there is one access type that has the highest possibility to contain the first detected fault. Workload 1 starts with workload block 1, whereas workload 2 starts with workload block 6 followed by block 1. The number of detected first faults directly correlates to this ordering. This effect is shown in table 5.6. For every workload block, it lists the ordering of the different workloads w.r.t. to the percentage of all detected first faults during execution of the block. For example, while executing workload 1 41.68% of all first faults detected during operations of workload block 1 were reported.

The cells highlighted with a light gray are cases were the result slightly differs from the expected behavior. However, the differences in the percentage are relatively small and only neighboring workloads in the workload block changed places. Thus, due to the random nature of the fault injection, these slight differences may not allow any reasoning about the failure rate of the different workload blocks. Furthermore, the workload blocks were designed to be self contained and thus do not depend on the previously executed blocks. This is confirmed by the fact that the failure rates mostly depend on the offset from the simulation start. Nevertheless, the bias due to the different workloads has to be taken into account at the evaluation of the failure rate of the workload blocks.

5.6.2 Influence of the Workload Blocks

The different workload blocks target to test different functionalities, described in section 5.5.1, of the MMU and are thus expected to cause different failure rates. Nevertheless, as mentioned in section 5.6.1, the number of detected first faults for each workload block depends mostly on its position within the workload. Thus, an evaluation of the different workload blocks w.r.t. the workloads is inadequate.

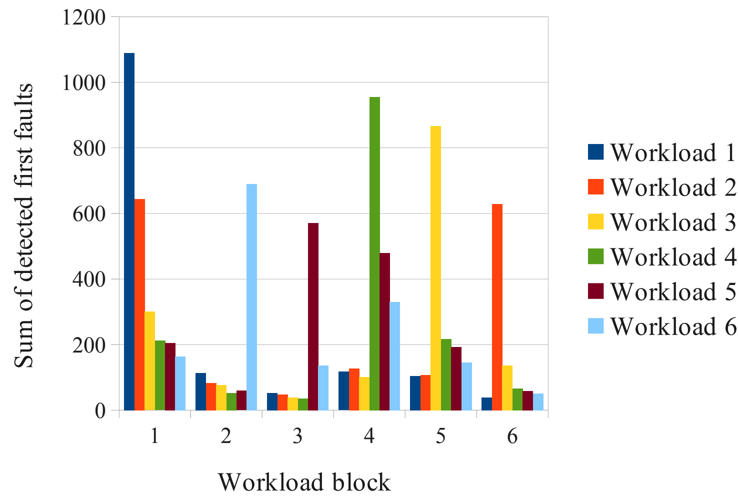

Figure 5.14: Sum of detected first faults w.r.t. workloads and workload blocks

1st block	2nd block	3rd block	4th block	5th block	6th block
workload: 1 expected: 1 (41.68 %)	workload: 6 expected: 6 (64.09 %)	workload: 5 expected: 5 (64.81 %)	workload: 4 expected: 4 (45.28 %)	workload: 3 expected: 3 (53.09 %)	workload: 2 expected: 2 (64.25 %)
workload: 2 expected: 2 (24.65 %)	workload: 1 expected: 1 (10.60 %)	workload: 6 expected: 6 (15.44 %)	workload: 5 expected: 5 (22.71 %)	workload: 4 expected: 4 (13.29 %)	workload: 3 expected: 3 (13.89 %)
workload: 3 expected: 3 (11.48 %)	workload: 2 expected: 2 (7.72 %)	workload: 1 expected: 1 (6.02 %)	workload: 6 expected: 6 (15.65 %)	workload: 5 expected: 5 (11.82 %)	workload: 4 expected: 4 (6.74 %)
workload: 4 expected: 4 (8.11 %)	workload: 3 expected: 3 (7.16 %)	workload: 2 expected: 2 (5.45 %)	workload: 2 expected: 1 (5.55 %)	workload: 6 expected: 6 (8.88 %)	workload: 5 expected: 5 (5.92 %)
workload: 5 expected: 5 (7.81 %)	workload: 5 expected: 4 (5.49 %)	workload: 3 expected: 3 (4.31 %)	workload: 1 expected: 2 (6.02 %)	workload: 2 expected: 1 (6.55 %)	workload: 6 expected: 6 (5.21 %)
workload: 6 expected: 6 (6.28 %)	workload: 4 expected: 5 (4.93 %)	workload: 4 expected: 4 (3.97 %)	workload: 3 expected: 3 (4.79 %)	workload: 1 expected: 2 (6.37 %)	workload: 1 expected: 1 (3.98 %)

Table 5.6: Workloads with the most detected first faults per workload block versus the expected ordering introduced by the workload block occurrences in the workloads

Workload	1	2	3	4	5	6
Workload block 1	3708.23 ns	3768.84 ns	3768.84 ns	3768.84 ns	3768.84 ns	3695.02 ns
Workload block 2	1542.80 ns	1542.80 ns	1542.80 ns	1542.80 ns	1534.46 ns	1934.01 ns
Workload block 3	1190.16 ns	1190.16 ns	1190.16 ns	1181.82 ns	1195.67 ns	1190.16 ns
Workload block 4	2490.52 ns	2490.52 ns	2482.18 ns	2496.03 ns	2490.52 ns	2490.52 ns
Workload block 5	2226.04 ns	2152.22 ns	2231.55 ns	2226.04 ns	2226.04 ns	2226.04 ns
Workload block 6	873.26 ns	931.19 ns	925.68 ns	925.68 ns	925.68 ns	925.68 ns

Table 5.7: Simulation duration of each workload block for the different workloads

	simulation duration rank	failure rate rank by sum	failure rate rank as first block
Workload block 1	1 (30.80 % / 29.67 %)	1 (28.13 %)	1 (22.69 %)
Workload block 2	4 (13.21 % / 15.48 %)	4 (11.57 %)	4 (14.35 %)
Workload block 3	5 (9.78 % / 9.57 %)	6 (9.48 %)	5 (11.9 %)
Workload block 4	2 (20.47 % / 19.97 %)	2 (22.70 %)	2 (19.90 %)
Workload block 5	3 (18.20 % / 17.86 %)	3 (17.58 %)	3 (18.06 %)
Workload block 6	6 (7.54 % / 7.45 %)	5 (10.54 %)	6 (13.10 %)

Table 5.8: Comparison of simulation durations versus failure rates

As mentioned in section 5.5.2, the different workloads vary slightly in their simulation duration due to the necessity of intermediate actions at the boundaries of the workload blocks, e.g., TLB flushes or table walks. Furthermore, the last workload block for each workload is slightly shorter because no TLB flush is needed afterwards. The duration for each workload block w.r.t. to each workload variation is shown in table 5.7.

These simulation durations may influence the number of detected first faults for each workload block. Table 5.8 lists in its second column the ordering of the workload blocks w.r.t. their simulation duration. The first number in brackets shows the percentage of the average complete simulation duration over all workloads for the block, whereas the second percentage number is w.r.t. to the sum of simulation durations when each block is the first block. This sum is generated by accumulating the simulation duration of block 1 in workload 1 with the simulation duration of block 6 in workload 2 and so on.

The third column ranks the workload blocks by their failure rate based on the sum over all workloads. This shows a clear correlation between failure rate and simulation duration. Only workload block 3 and 6 differ slightly from the expected behavior. The percentage numbers should be compared to the first numbers in column 2.

In contrast, the fourth column is based on the sum over the workloads where each block is the first executed block. Thus the base for the percentage calculation is generated in the same way as the second number in the first column but w.r.t. the number of detected first faults.

As it can be seen, the percentage numbers correlate strongly with the corresponding simula-

tion duration percentages but still differ up to 6.98 % (workload block 1 as first block w.r.t. to its first block simulation duration). Thus, we can conclude that the simulation duration matters a lot but does not completely explain the failure rates. A possibility for future work would be to trim all workload blocks to the same simulation duration and evaluate their failure rates to analyze this in more detail. Nevertheless, for the above mentioned reasons we decided to perform our analysis on the indicated MMU operations over all workloads and not w.r.t. the workload blocks.

5.6.3 Detected First Faults per Indicated Operation

Due to the wide influence of the workload parameters described in the previous sections, we decided not to evaluate the detected first faults based on the workload blocks. Instead, we evaluate the different operations at the MMU by the reported codes of the test bench and the traced signal values.

The reported codes for the different signal states were already presented in section 5.4.1. In order to keep the evaluation readable, we use the same codes. Note that these codes do not cover differences in data values, they correspond only to the performed action.

Furthermore, if there is more than one action code reported in the same clock cycle for the tested MMU, each of them is counted as one detected first deviation for the sum of detected first faults. Due to the reasons described in section 5.4.2, the comparison for the current trace file is stopped after the first deviating action codes.

Table 5.9 shows the absolute number of detected first deviations from the intended action code (the first column) due to the fault injection. The intended action is determined by the reference MMU, where each legal state corresponds to exactly one action code at the same rising clock edge. Nevertheless, the tested MMU may report multiple performed action codes at a time. Thus, it is possible that the reference MMU reports code D1 while the tested MMU reports codes D1 and M3. This would increment the counters in row D1 at column D1 and M3 by one and corresponds to the case when a data translation result is indicated with the instruction translation result indicator containing a meta-value at the same rising clock edge. The numbers highlighted with a dark gray show the highest count, whereas, the light gray background highlights numbers with 3 digits. This was chosen to emphasize the highest counts. Furthermore, the cells with a medium gray show cases where the intended action equals the performed action but an additional action is reported.

Some action codes are not expected to be intended codes and thus are not included as a separate row in the table. The evaluation has shown that such codes indeed are never reported for the reference MMU. This applies to the codes that report a meta-value or a timeout. Furthermore, code M5 was never reported for both MMUs, as the corresponding signal is driven by the test bench and thus cannot contain a meta-value.

performed action	D1	D2	I1	I2	FS	MR	MW	R	T	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	
D1	2	51	23	9	7	68	18	0	32	29	100	34	29	0	0	18	0	27	26	473
D2	78	2	19	2	11	62	35	0	43	11	57	21	13	0	0	32	3	9	9	407
I1	4	1	0	15	3	47	6	0	2	10	10	32	10	0	0	0	4	9	9	162
I2	21	4	35	3	12	103	11	0	3	23	17	56	18	0	0	1	16	15	16	354
FS	9	17	11	18	43	71	8	0	6	26	23	25	24	0	0	0	0	16	16	313
MR	56	29	27	6	29	32	45	132	478	164	126	96	94	0	11	0	0	83	84	1492
MW	31	39	7	0	12	298	11	7	76	65	96	24	22	0	3	1	0	21	21	734
R	9	3	3	0	11	39	31	60	0	67	44	41	49	0	0	0	0	42	41	440
intended action																				sum

Table 5.9: Number of deviations of the performed actions w.r.t. to the intended actions

intended action	evaluation count	evaluation rank	deviation count	deviation rank
D1	709426	6 (5.67 %)	473	3 (10.81 %)
D2	710339	5 (5.68 %)	407	5 (9.30 %)
I1	253829	8 (2.03 %)	162	8 (3.70 %)
I2	306406	7 (2.45 %)	354	6 (8.09 %)
FS	3045210	2 (24.35 %)	313	7 (7.15 %)
MR	3812534	1 (30.48 %)	1492	1 (34.10 %)
MW	1177383	4 (9.41 %)	734	2 (16.78 %)
R	2492482	3 (19.93 %)	440	4 (10.06 %)

Table 5.10: Weak correlation between how often an intended action code is evaluated and how often a deviation is detected

Table 5.10 shows the total count of how often each intended action code was evaluated by the comparison GUI. Note that this does not reflect the absolute count of how often each action was actually performed by the MMU, because the comparison stops after the first deviation in each trace file.

The deviation rank in table 5.10 shows no direct correlation to the evaluation count. Thus, the number of deviations is further dependent on other factors, like, e.g., the simulation duration of each intended action. Nevertheless, the actual deviation rate of a real system is strongly dependent on technology factors and thus this simulated fault injection does not allow to draw in depth conclusions about it. Hence, we decided to state these numbers as is and only give a short tendency explanation.

As it can be seen in table 5.10 the numbers of deviations related to the memory interface, i.e., MR, MW and R, clearly exceed the other values. This is due to the fact that memory related operations usually take multiple clock cycles and therefore their window of vulnerability is bigger. For example, our simulated memory interface in the test bench performs a memory lookup within 3 clock cycles. This is much faster than a real system might handle it and was chosen to cut down the simulation time. Thus, for a real system the window of vulnerability will be much wider. Nevertheless, our workloads make heavy use of memory operations due to the small TLB and the testing of different functionalities. In a real system a table walk is more a rare case than an usual one. However, if a memory read or write operation fails, without being detected, it may influence many following translations, whereas a single deviation in a translation response may only affect one memory access. Therefore, we consider the memory interface as the most critical path of the MMU.

5.6.4 Special Cases due to the Fault Injection

The following two paragraphs discuss special cases that were caused by the fault injection but have to be handled outside the MMU.

5.6.4.1 Meta-values at the Signals

The behavior in case of meta-values is dependent on the implementation of the evaluating circuit and thus no reasoning about the possible consequences is performed. Nevertheless, all indicator signals that show meta-values during the fault injection are included in the statistics and meta-values at data signals are considered as wrong but valid data.

Furthermore, values of 'X', i.e., "forcing unknown", may be caused by driver conflicts and can potentially lead to a permanent fault in a hardware realization. The number of detected first faults with a value of 'X' for all indicator and data signals was 1380 which corresponds to 14.85 % of all detected first faults. Thus, this type of fault cannot be considered a rare case.

5.6.4.2 Timeouts

In case the tested MMU did not respond to the issued translation request within 80 clock cycles, the test bench reports a timeout. If the MMU did respond with another action code than expected, this deviation is detected instead of the timeout. Furthermore, if the tested MMU responds with data that deviates from the reference MMU this is also detected instead of the timeout. Timeouts were reported by the performed action code T and the number of detected timeouts was 640.

The behavior in case of a timeout is dependent on the implementation of the evaluating circuit. Thus, no discussion about its consequences can be given in the limited context of this thesis.

5.6.5 Consequences of Deviating Actions

The following section is going to evaluate the possible consequences, that may be caused if the tested MMU deviates from the reference MMU by the reported action codes. Furthermore, the consequences are mapped to the failure effects of the theoretical analysis in chapter 4.

Nevertheless, this description is purely based on the indicator signals and thus does not deal with wrong translation data. This is going to be analyzed in section 5.6.6.

5.6.5.1 Valid Indicator Signals

In case the indicator signals contain a valid but unexpected state, this type of deviation is detected by the result comparison GUI and thus can be further evaluated. The following enumeration deals with the various cases and describes possible consequences. Furthermore, table 5.16 provides an immediate mapping without additional explanations.

1. Deviations from action code D1 which represents a data translation result with no access exception flagged:

   D1: The intended action code is active together with another code.

   D2: A data translation result with an access exception is flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

   I1: An instruction translation result with no access exception is flagged instead or together with a data translation result. Corresponding fault type at theoretical analy-

sis: No separation between instruction and data translations at theoretical analysis performed.

I2: An instruction translation result with an access exception is flagged instead or together with a data translation result. Corresponding fault type at theoretical analysis: No separation between instruction and data translations at theoretical analysis performed.

FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

MR: A table walk read operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 13

MW: A table walk write operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 12

R: This was never caused by the fault injection.

2. Deviations from action code D2 which represents a data translation result with an access exception flagged:

D1: A data translation result with no access exception is flagged although an access exception was expected. Corresponding fault type at theoretical analysis: FO-FH 2

D2: The intended action code is active together with another code.

I1: An instruction translation result with no access exception is flagged. No separation between instruction and data translations at theoretical analysis performed.

I2: An instruction translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: No separation between instruction and data translations at theoretical analysis performed.

FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

MR: A table walk read operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 13

MW: A table walk write operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 12

R: This was never caused by the fault injection.

3. Deviations from action code I1 which represents an instruction translation result with no access exception flagged:

D1: A data translation result with no access exception is flagged. No separation between instruction and data translations at theoretical analysis performed.

D2: A data translation result with an access exception is flagged. No separation between instruction and data translations at theoretical analysis performed.

I1: The intended action code is active together with another code. This was never caused by the fault injection.

I2: An instruction translation result with an access exception is flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

MR: A table walk read operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 13

MW: A table walk write operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 12

R: This was never caused by the fault injection.

4. Deviations from action code I2 which represents an instruction translation result with an access exception flagged:

D1: A data translation result with no access exception is flagged. No separation between instruction and data translations at theoretical analysis performed.

D2: A data translation result with an access exception is flagged. No separation between instruction and data translations at theoretical analysis performed.

I1: An Instruction translation result with no access exception is flagged although an access exception was expected. Corresponding fault type at theoretical analysis: FO-FH 2

I2: The intended action code is active together with another code.

FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

MR: A table walk read operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 13

MW: A table walk write operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 12

R: This was never caused by the fault injection.

5. Deviations from action code FS which represents a new fault in the fault status register flagged:

D1: A data translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FO-FH 2

D2: A data translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: T-FH 3

I1: An Instruction translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FO-FH 2

I2: An instruction translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: T-FH 3

FS: The intended action code is active together with another code.

MR: A table walk read operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 13

MW: A table walk write operation is performed, although none was expected. Corresponding fault type at theoretical analysis: T-HTW 12

R: This was never caused by the fault injection.

6. Deviations from action code MR which represents a table walk read operation:

   - D1: A data translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: T-HTW 11
   - D2: A data translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: T-HTW 11
   - I1: An Instruction translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: T-HTW 11
   - I2: An instruction translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: T-HTW 11
   - FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1
   - MR: The intended action code is active together with another code.
   - MW: A table walk write operation is performed instead of a table walk read operation. Corresponding fault type at theoretical analysis: FO-HTW 14
   - R: A table walk result is indicated although none was expected. This happens if a table walk operation is interrupted and is described in more detail in section 5.6.5.2.

7. Deviations from action code MW which represents a table walk write operation:

   - D1: A data translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10
   - D2: A data translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10
   - I1: An Instruction translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10
   - I2: An instruction translation result with an access exception is flagged. This was never caused by the fault injection.
   - FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1
   - MR: A table walk read operation is performed instead of a table walk write operation. Corresponding fault type at theoretical analysis: FA-HTW 15
   - MW: The intended action code is active together with another code.
   - R: A table walk result is indicated although none was expected. This happens if a table walk operation is interrupted and is described in more detail in section 5.6.5.2.

8. Deviations from action code R which represents a table walk response:

   - D1: A data translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10 or T-HTW 11
   - D2: A data translation result with an access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10 or T-HTW 11
   - I1: An Instruction translation result with no access exception is flagged. Corresponding fault type at theoretical analysis: FA-HTW 10 or T-HTW 11

I2: An instruction translation result with an access exception is flagged. This was never caused by the fault injection.

FS: A new fault in the fault status register flagged, although none was expected. Corresponding fault type at theoretical analysis: T-FH 1

MR: A table walk read operation is still performed although already a result is expected. This is caused by misbehavior of the memory read request and its effects are implementation dependent, hence no mapping is provided.

MW: A table walk write operation is still performed although already a result is expected. This is caused by misbehavior of the memory write request and its effects are implementation dependent, hence no mapping is provided.

R: The intended action code is active together with another code.

5.6.5.2 Table Walk Response Instead of Table Walk Request

The fault injection consequence of a table walk response instead of a table walk request at first may seem counterintuitive. This effect is caused by the implementation of the emulated memory at the test bench. It can be seen in table 5.9 at column R of the rows MR and MW.

The cause of this effect is that the MMU has to issue the memory request until the test bench acknowledges the request. In case the request is released in the same clock cycle when it is acknowledged, the test bench later on responds with data according to the signal values that were read during the acknowledging clock cycle. If the MMU releases the request, the signal that indicates a read or write request may indicate a write request to address zero, i.e., its default state. Hence, the test bench acknowledges this write request and the trace file contains a memory response instead of the released memory request. This is shown in figure 5.15 and may lead to the failure effects FO-HTW 5 or FO-HTW 9. In fact, this type of fault was only observed two times during the complete fault injection.

A real implementation of a memory interface may behave differently, but memory requests are expected to take more than one clock cycle and thus the observed behavior is considered an error with a failure as consequence. Furthermore, the test bench implementation is conform with the description of the memory interface in [Eis02, p. 20].

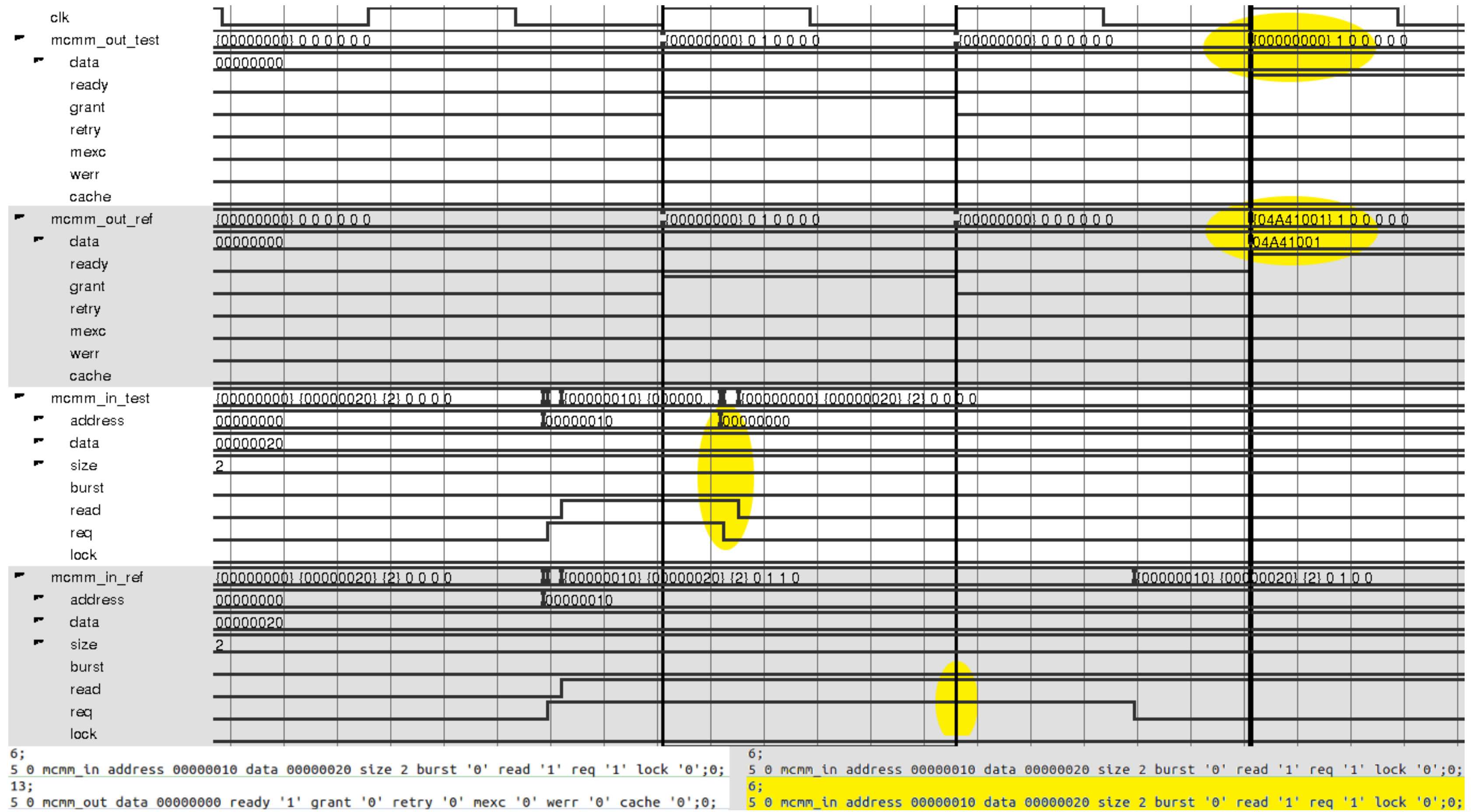

Figure 5.15: Example waveform of a difference at the *mcmm_in* and *mcmm_out* signals

5.6.6 Consequences of Deviating Data

In case the intended and the performed action codes do not differ, there is still the possibility that the data signals of the tested MMU contain different values than the reference MMU. The following section deals with these cases.

Note that in case there are multiple results available in the same clock cycle, the result comparison GUI filters the list for the one indicated by the reference MMU and compares the data values of these results. Thus, also these cases are included in the following discussion.

5.6.6.1 Differences at the Output to Memory

The output interface to memory consists of 3 signal vectors *address*, *data* and *size* together with 4 single signals *burst*, *lock*, *read* and *req*. The *address* vector corresponds to the accessed memory location and is used for both, reads and writes. The *data* vector in turn is only used for writes to memory and contains the data to be written. The *size*, *burst* and *lock* signals are AHB related and are set to constant values within the source code of the LEON3 MMU. The *read* signals differentiates between a memory write and read transfer and the *req* signal initializes the transfer. The last two signals are already covered by the evaluation of the indicated operations and thus are excluded at this point.

During read requests, i.e., action code MR, the numbers of differences shown in table 5.11 were gathered. A difference in the address corresponds to FO-MI 4 or T-MI 6 and may cause severe consequences later on. The differences at the data signal vector do not matter during read operations, because this signal vector is an output of the MMU and is not expected to be evaluated during reads. Additionally, the differences at the AHB related signals are dependent on the memory interface implementation. The emulated memory interface did not handle these signals.

Table 5.12 shows the numbers of differences during write requests, i.e., action code MW in relation to the detected first faults at all data signals. At this point, a difference in the address corresponds to FO-MI 3 or T-MI 5. Due to the fact that this is a write operation, the consequences can be catastrophic because the data may be written at any place in memory, possibly crashing the whole system. A difference at the data signal vector corresponds to FO-MI 1 and FO-MI 2, with the same severe consequences, if the data is used later on.

5.6.6.2 Differences at the Translation Outputs

Table 5.13 shows the number of differences at the data translation results in relation to the detected first faults at all data signals. The *data* signal represents the complete translation result and differences at this signal may correspond to all AM block failure effects from the theoretical analysis in section 4.2.9. These are FO-AM 1, FO-AM 2, FO-AM 3, FA-AM 4, FO-AM 5 and FA-AM 6. All of them may have severe consequences.

The same analysis holds true for the number of differences at the instruction translation results in table 5.14, although instruction translations are in principle not covered by the theoretical analysis. Both translation result types also show a number of differences at the signal *cache* that determines whether the accessed memory region is cacheable or not. This signal is used for the

signal	count
address	1177 (23.95 %)
data	487 (9.91 %)
size	98 (1.99 %)
burst	34 (0.69 %)
read	0 (0.00 %)
req	0 (0.00 %)
lock	35 (0.71 %)
sum	1831 (37.25 %)

Table 5.11: Numbers of differences during memory read request

signal	count
address	302 (6.14 %)
data	602 (12.25 %)
size	23 (0.47 %)
burst	9 (0.18 %)
read	0 (0.00 %)
req	0 (0.00 %)
lock	9 (0.18 %)
sum	945 (19.23 %)

Table 5.12: Numbers of differences during memory write request

signal	count
data	898 (18.27 %)
cache	33 (0.67 %)
accexc	0 (0.00 %)
sum	931 (18.94 %)

Table 5.13: Numbers of differences at data translation results

signal	count
data	269 (5.47 %)
cache	16 (0.33 %)
accexc	0 (0.00 %)
sum	285 (5.80 %)

Table 5.14: Numbers of differences at instruction translation results

implementation of memory mapped IO. Nevertheless, this feature is not covered in the theoretical analysis and thus cannot be mapped to a corresponding failure effect. The access exception signal *accexc* is already covered by the analysis of the action codes.

5.6.6.3 Differences at the Fault Status Register

The effects of differences in the fault status register are implementation dependent on the OS. At the level of the MMU all of them (table 5.15) correspond to T-FH 3 because the cases where an unexpected fault or no fault is reported, are already covered by the analysis of the action codes. The meaning of the different signals can be found in [Eis02, p. 30f] and the percentages are in relation to the detected first faults at all data signals.

5.7 Results and Conclusion

The fault injection did not reveal additional failure effects to the theoretical analysis. Nevertheless, it pointed out the criticality of the memory interface and that the original consideration as a intermediate failure effect is not sufficient. Hence, the failure effect FO-MI 3 at the MI block should also be considered an output failure. The analysis was adapted to cover this experienced behavior.

signal	count
fs.ow	25 (0.51 %)
fs.fav	42 (0.85 %)
fs.ft	130 (2.64 %)
fs.at_ls	85 (1.73 %)
fs.at_id	35 (0.71 %)
fs.at_su	52 (1.06 %)
fs.l	92 (1.87 %)
fs.fa	460 (9.36 %)
sum	921 (18.74 %)

Table 5.15: Numbers of differences at the fault status register

Furthermore, some intermediate failures were noticeable by their effects at the memory interface of the LEON3 MMU. These were the failure effects FO-MI 1, FO-MI 2, FO-MI 4, T-MI 5, T-MI 6, FO-HTW 5, FO-HTW 9, FA-HTW 10, T-HTW 11, T-HTW 12, T-HTW 13 and FO-HTW 14.

Table 5.16 provides a mapping from the deviations between the intended and the performed action codes to the failure effects of the theoretical analysis. Table 5.17 provides the same for the differences in the signal values. This mapping allows to map the detected first differences to the failure effects, and table 5.18 shows the final result of the fault injection experiment. The numbers for the theoretical blocks should not be summed up without the mapping in mind. The last column provides the corresponding total number for all blocks considering the mapping. Note that the number of detected first differences in this summary does not add up to the total number of detected first differences, because not all of them have been mapped to the theoretical analysis due to the non-implemented features in chapter 4 as listed in table 5.2. Nevertheless, the percentages in brackets are w.r.t. to the sum of all detected first differences and timeouts.

Finally, we can conclude that the evaluation of the fault injection has allowed us to verify the theoretical analysis in chapter 4 by a mapping of the experienced failure effects to the theoretically described ones. Furthermore, it pointed out that memory interface is a critical part, and allowed us to improve the theoretical analysis.

performed action	D1	D2	I1	I2	FS	MR	MW	R
D1		T-FH 1			T-FH 1	T-HTW 13	T-HTW 12	
D2	FO-FH 2				T-FH 1	T-HTW 13	T-HTW 12	
I1				T-FH 1	T-FH 1	T-HTW 13	T-HTW 12	
I2			FO-FH 2		T-FH 1	T-HTW 13	T-HTW 12	
FS	FO-FH 2	T-FH 3	FO-FH 2	T-FH 3		T-HTW 13	T-HTW 12	
MR	T-HTW 11	T-HTW 11	T-HTW 11	T-HTW 11	T-FH 1		FO-HTW 14	FO-HTW 5, FO-HTW 9
MW	FA-HTW 10	FA-HTW 10	FA-HTW 10		T-FH 1	FA-HTW 15		FO-HTW 5, FO-HTW 9
R	FA-HTW 10, T-HTW 11	FA-HTW 10, T-HTW 11	FA-HTW 10, T-HTW 11		T-FH 1			
intended action								

Table 5.16: Mapping of deviating actions to failure effects

Signal	Failure effect
Address of memory read	FO-MI 4, T-MI 6
Address of memory write	FO-MI 3, T-MI 5
Data of memory write	FO-MI 1, FO-MI 2
Data of data translation result	FO-AM 1, FO-AM 2, FO-AM 3, FA-AM 4, FO-AM 5, FA-AM 6
Data of instruction translation result	FO-AM 1, FO-AM 2, FO-AM 3, FA-AM 4, FO-AM 5, FA-AM 6

Table 5.17: Mapping of deviating signal values to failure effects

Theoretical block	Failure effect	Output failure	Number of differences	Sum
FH	T-FH 1	Yes	151	1240 (13.35 %)
	FO-FH 2	Yes	133	
	T-FH 3	Yes	956	
AM	FO-AM 1	Yes	1167	1167 (12.56 %)
	FO-AM 2	Yes	1167	
	FO-AM 3	Yes	1167	
	FA-AM 4	Yes	1167	
	FO-AM 5	Yes	1167	
	FA-AM 6	Yes	1167	
MI	FO-MI 1	No	602	2979 (32.07 %)
	FO-MI 2	No	898	
	FO-MI 3	Yes	302	
	FO-MI 4	No	1177	
	T-MI 5	No	302	
	T-MI 6	No	1177	
HTW	FO-HTW 5	No	139	1050 (11.3 %)
	FO-HTW 9	No	139	
	FA-HTW 10	No	92	
	T-HTW 11	No	133	
	T-HTW 12	No	78	
	T-HTW 13	No	280	
	FO-HTW 14	No	45	
	FA-HTW 15	No	298	

Table 5.18: Number of differences mapped to the failure effects of the theoretical analysis

CHAPTER 6

Safety Enhancement Possibilities for Memory Management Unit

This chapter provides an overview of different possibilities for error mitigation and proposes relevant mechanisms for future implementations of safety enhanced MMUs that detect and indicate failures with a very high probability, but may not provide error correction capabilities, to allow usage in fail-safe systems.

6.1 Possibilities for Error Mitigation

There are various possibilities to detect errors and some of them additionally allow error correction. However, all of them come with their own specific additional costs, therefore it is critical to select the most suitable mitigation mechanism to gain an optimal result. The following section describes the basics of different types of mitigation mechanisms and their tradeoffs.

6.1.1 Parity

One technique to detect errors in memories and signal vectors is to add a parity bit. This allows to detect single errors but does not provide sufficient information to correct the errors. However, in case of a MMU this protection mechanism may be sufficient to handle transient faults. If a TLB entry gets corrupted, the entry may be marked as flushed from the TLB and therefore a translation using the entry results in a page table walk. In case of an error at a signal vector, the whole translation process may be restarted. Nevertheless, both possibilities result in a delayed translation and therefore have implications at real-time systems. Furthermore, parity can only be used on memory elements and signal vectors. To use parity for error detection of logical functions, the function needs to allow parity prediction, which is not feasible for complex functions.

6.1.2 Dual Rail Encoding

This technique is often used for the construction of speed independent asynchronous circuits. Nevertheless, dual rail encoding, which actually represents a 1-of-2 encoding, can also be used to protect a signal line against a single error. A bit is represented by two signal lines, where one holds the inverted value of the other, thus the correct codewords are (01) and (10). Whenever a value of (11) or (00) is detected an error occurred. However, dual rail encoding does not allow to correct this error.[WW05]

6.1.3 Single-Error Correction and Double-Error Detection (SEC-DED)

A SEC-DED Error-Correcting Code (ECC) may be used to correct a single error in a memory element or a signal vector. In order to protect a word of n bits, k check bits have to be added, where $k = \lceil log2(n + k + 1) \rceil$[Nic05, p. 406]. The correction of CAM entries, which are used in some TLBs, comes with one complication:

"*When an error affects the address stored in a CAM word, an associative search of the correct address value will produce an incorrect miss. This error cannot be detected since the word containing the erroneous address is not read to check for its integrity. As a matter of fact, we can use parity or ECC codes to protect the data field and the address field of CAMs. With this solution, the data field will be completely protected. However, the address field can be protected only when the error leads to an incorrect hit since in this case the hit word can be read and checked, but it cannot be protected when the error leads to an incorrect miss. An incorrect miss will not be a problem in systems where CAM is used as a cache memory since, in this case, the data will be retrieved from the main memory. But in numerous systems employing CAMs, this is not the case.*" [Nic05, p. 409]

A possibility to overcome this problem is, e.g., given in [HGS05]. Nevertheless, it may be the case that SEC-DED is not worth the overhead compared to parity for TLB entries or signal vectors within a MMU.

6.1.4 SRAM Protection through Built-In Current Sensors (BICSs)

A SEU induced bit flip at a SRAM cell causes a transient current flow through one of the power supply lines. Thus, a BICS can be used to sense this abnormal current in order to detect bit flips. By using BICSs to monitor the vertical power lines of SRAM cells and adding a parity bit to every memory word, it is possible to locate and correct a single bit flip within the memory. The main advantage of this technique is its low area cost:

"*The technique is promising since it requires a very low area cost of 4% to 7% according to each specific implementation and is very suitable to protect memories that are difficult to protect by ECC, such as memories using maskable operations or CAMs.*" [Nic05, p. 410]

Furthermore, if combined with SEC-DED ECC, BICSs can also be used to detect and correct Single-word Multiple-bit Upsets (SMUs) in SRAM [GNP05].

6.1.5 Hardened Latches/Flip-flops

In order to reduce the susceptibility against SEE of logic parts, another solution is to use hardened latches and flip-flops. However, some of the hardening techniques require a specific transistor size and thus do not scale easily with shrinking device structures. An alternative is the Dual Interlocked Storage Cell (DICE)[CNV96], which scales but comes with a relatively big area overhead.

"*So, the DICE cell [sic] can be used to protect SRAMs, register files, CAMs, the configuration memory of FPGAs, and the latches or flip-flops of logic designs. Comparison between industrial flip-flop and DICE-based flip-flop performed for a 90-nm process node shows 81% area overhead, identical rise time (0.13 ns in both case), and a 20% fall time increase (0.12 ns versus 0.10 ns).*" [Nic05, p. 411]

6.1.6 Hardware Redundancy

Duplication with comparison or Triple Modular Redundancy (TMR) comes with a huge area and power penalty but allows to detect and in case of TMR correct SEE even in logic parts. However, due to the overhead the usage of this techniques is "*unacceptable in most designs*" [Nic05, p. 406]. Nevertheless, it is relatively easy to achieve and can also be used on parts of the system, where other mitigation strategies are hard to implement. There is also tool support for automatic implementation of TMR on the basis of an existing design, e.g., [Xil12].

6.1.7 State Machine Protection

Faults may affect the state sequence of a control circuit. A possible technique to allow detection of deviation from the intended behavior is to check the signature of a group of sequential states against predefined possible signatures. This allows runtime detection of state errors.[EME94]

6.1.8 Time Redundancy Implemented in Software

Another possibility is to execute an operation multiple times and compare the result. However, this comes with a high memory overhead and speed penalty. It is the latter that makes this technique impractical for a performance related operation like the translation of virtual addresses by the MMU.

"*The interest of the software-implemented approach is its flexibility since it does not require any hardware modification. The counterpart is a high memory overhead (5x for error detection, 6x for error detection and recovery) and speed penalty (3x for error detection, 4x for error detection and recovery).*" [Nic05, p. 412]

6.2 Relevant Mechanisms for the Implementation of a Safety Enhanced MMU

A MMU typically introduces a timing uncertainty at memory accesses (see section 2.2.6). Hence, the system must be capable to deal with a TLB miss or a page fault. As the latter already intro-

duces a huge timing overhead, we conclude that it is safe to let a failure in the MMU introduce an overhead of about the same range. Therefore, we propose to implement the MMU in a way, such that a single error is detected and a corresponding error signal is raised. Hence, the OS or specialized hardware can take care of the problem. Possible responses are, e.g., retry of the failing translation, emergency shutdown or deactivation of the MMU.

In case the system has to fulfill stricter real-time requirements, the MMU has to provide a possibility for TLB entry locking. In case of a TLB that uses CAMs special care has to be taken, like for example [HGS05], to prevent false misses which could violate the timing requirements. Direct mapped TLBs can be easier protected by, e.g., parity bits.

The main goal of this chapter is to establish the fundamentals for the development of safety enhanced MMUs that detect and indicate failures with a very high probability to allow usage in fail-safe systems. Therefore, it is highly recommended to prohibit faulty memory accesses caused by MMU failures but no error correction within the MMU is needed. Due to the main focus on the MMU hardware, the following mechanisms are considered relevant:

- Parity
- Dual Rail Encoding
- Hardware Redundancy

Furthermore, the protection mechanisms used for space applications, e.g., radiation hardened devices, are expected to perform well on ground level. Nevertheless, an in depth evaluation of this mechanisms is out of focus of this work and we restrict ourself to mechanisms that are implementable in commercial FPGAs.

6.3 General approach for Protection of a MMU

The following discussion on the implementation possibilities of protection mechanisms for MMUs is evaluated at the example of the LEON3 MMU. Selection of appropriate protection mechanisms is a tradeoff between engineering costs for the mechanism and hardware costs of the modules that have to be protected. For small but complex modules the effort on complicated protection mechanisms may not pay off and thus the mechanism with the lowest engineering cost, like, e.g., duplication, may be chosen.

The hardware costs of the different LEON3 MMU modules are shown in table 6.1. The two modules with the biggest numbers of combinationals and registers are the ones called “various logic” and the CAM modules. The first one was introduced to summarize the resource usage that cannot be mapped to separate modules. Another part with a high hardware count is the LRU module. However, in hardware cost sensitive applications, this module can be replaced by a simple counter.

In order to show the tendency for increased performance, the hardware costs were analyzed for a LEON3 MMU with 4 TLB entries as well as for 32 TLB entries. As it can be seen, the HTW module is the only module that is constant in its hardware count.

Module	TLB Entries	Logic Cell Combinationals	Logic Cell Registers	Memory Bits
LRU Module	4	37 (5.75 %)	15 (3.17 %)	
CAM Modules	4	138 (21.46 %)	252 (53.28 %)	
RAM	4			120 (100 %)
HTW Module	4	99 (15.4 %)	70 (14.8 %)	
Various Logic	4	369 (57.39 %)	136 (28.75 %)	
Complete MMU	4	643	473	120
LRU Module	32	603 (21 %)	195 (8.05 %)	
CAM Modules	32	1031 (35.9 %)	2016 (83.2 %)	
RAM	32			960 (100 %)
HTW Module	32	99 (3.45 %)	70 (2.89 %)	
Various Logic	32	1140 (39.68 %)	142 (5.86 %)	
Complete MMU	32	2873	2423	960

Table 6.1: Size of LEON3 Modules

6.4 Analysis of Protection Approaches for the LEON3 MMU

The following section focuses on a theoretical analysis of different protection possibilities for the LEON3 MMU. Each of the possibilities comes with its already explored advantages and drawbacks. The analysis of an actual implementation may not lead to new findings and thus was skipped for some of the approaches.

6.4.1 Protection Approach with Duplication of the complete MMU besides the CAM Modules

One possibility is to duplicate every MMU module, besides the CAM module. In this implementation the CAMs have to be well protected against SEE, because they form a single point of failure. However, due to the complex functionality of the CAM a parity prediction circuit for the whole block is not an option. Furthermore, there are signals inside the CAMs that influence more than one output signal and thus may cause failures that are undetectable by parity checking. Hence, one approach is to combine internal signal vectors as far as possible and protect each of those combined signal vectors by parity. Also the input and output signals have to be protected by parity. Furthermore, logical functions, that cannot be protected by parity, have to be duplicated and compared. Nevertheless, a test implementation showed that this approach leads to a blow up of the CAMs that exceeds the simple duplication of the modules. Thus, this approach has no advantage over a complete duplication of the MMU and is challenging to implement.

6.4.2 Duplication of the Complete MMU

This is the simplest approach w.r.t. VHDL design complexity. Nevertheless, there are still a few weak points in this implementation. First of all, the MMU has to be seen in the context of a

complete system. Thus, if only the MMU is duplicated the input signals to the MMU may still be corrupted and in general, a protected MMU without a protected CPU may not be beneficial. Thus, we recommend to shift the problem at another level, by either using a dual core processor with two separate MMUs or two separate processors. However, we want to point out that the duplication of the MMU has two weak points. First of all, as already mentioned the inputs signals have to be correct and second, the comparison of the output values has to be performed in a safe way. This can be achieved by, e.g., using parity at both outputs of the duplicated circuits and a self-checking checker (see, e.g., [AM73]) at the output of the MMU.

6.4.3 Protection of the RAM Portion and Duplication of Everything else

As mentioned in [Eis02, p. 51] the RAM, that is used to store the data of TLB entries, could be replaced by registers for performance and code complexity optimizations. This would imply the same difficulties for their protection as already mentioned in 6.4.1. If the RAM elements are not replaced, they can be protected by a parity bit with the advantage of significant memory savings compared to duplication. Each of the duplicated logics has to calculate the parity of the data that is to be transmitted to the RAM. The data has to be compared at the input of the RAM. Due to the single fault assumption no parity check has to be performed at that point. The output data then has to be transmitted to both duplicated logics and a parity check has to be performed by both of them. Again, there is a safe output comparison needed for the outputs of the duplicated logic. This approach is considered to save some memory bits and to work well in practice.

6.4.4 Other Optimization Possibilities

In order to reduce the hardware effort for protection of the MMU one could restrict the protection to the parts that actually have the power to cause failures at the output. However, the only part of the MMU that may be considered to be left unprotected is the LRU module. In case of a fault at this module, the MMU may show the performance of a MMU with just a single TLB entry but no output failures occur. Nevertheless, if hardware cost is an issue, it may be worth considering to replace the complete LRU module by protected counters instead.

Another possibility is to restrict the protection mechanisms to parts of the MMU that are most frequently influenced by SEE. This requires measurements of real world data of hardware implementations and may be even susceptible to process variations at the production of the Integrated Circuits (ICs).

6.5 Conclusion

The evaluation of various protection mechanisms, that are implementable in commercial FPGAs, has shown that the mechanisms may not outperform the complete duplication of the MMU and thus, this protection level is not recommended. Furthermore, one has to keep in mind that the MMU cannot be seen as independent of the remaining system. Therefore, we would recommend to duplicate the complete processor if the targeted reliability is not met by the usage of single MMU.

CHAPTER 7

Results and Conclusion

This thesis has provided an overview of different memory management and protection mechanisms with an emphasis on virtual memory and the functionality of MMUs. Furthermore, a selection of modern processors was reviewed for their MMU features. A theoretical MMU was sketched and its possible failure modes were analyzed. The theoretical analysis was backed up by a simulated fault injection in the LEON3 MMU and the evaluation thereof. Afterwards a variety of possible mitigation mechanisms were introduced and their feasibility analyzed.

According to our results, a MMU may fail in an unsafe way and thus a critical evaluation of its failure rate has to be performed if usage in a safety critical environment is targeted. If the failure rate exceeds the tolerable limit for the given application, we suggest to use a dual core processor with two separate MMUs or two separate processors, depending on the targeted reliability.

A possibility for future work could be the verification of the gained results by a physical fault injection in a hardware realized MMU. It can be seen in this work that the MMU closely interacts with the processor and memory. Hence, we would suggest to enclose those units in the fault injection. This enables the usage of longer and more realistic workloads and may allow the fault injection to be less dependent on the implementation of the test bench.

APPENDIX A

E-Mail Correspondence

```
Dear Oliver,

The AMBA address space used in GRLIB is 4 GiB (32-bits) and the top most
bits of the physical page number are discarded within the processor. The
documentation is misleading and we will update it.

Best regards,
  Jan

On 11/14/12 4:57 PM, Oliver Hechinger wrote:
> Dear Sir or Madam,
>
> I am a student on computer engineering at the Vienna University of
> Technology (Austria). During my research on my master thesis about
> memory management units the following question about the MMU of LEON3 arose:
>
> According to the datasheet, LEON3 is capable of handling 36-bit physical
> addresses. Nevertheless, the output signals of the VHDL MMU entity
> (mmudco, mmuico with datatypes mmudc_out_type, mmuic_out_type) only
> contain 32-bit wide address buses. Furthermore, the TLB_MergeData
> procedure only uses 20 bits (PTE(27 downto 8)) of the page frame address
> stored at the page table entry (considering a hit at page level with 4k
> pages).
>
> How is it possible to generate 36-bit physical addresses under this
> conditions? What am I missing?
>
> I would really appreciate your help. Thanks in advance!
>
> Best regards,
> Oliver Hechinger
```

Acronyms

AM	Address Merger
AMBA	Advanced Microprocessor Bus Architecture
AHB	AMBA High-performance Bus
BICS	Built-In Current Sensor
CAM	Content Addressable Memory
CSV	Comma-Separated Values
CPI	Cycles per Instruction
CPLB	Cacheability Protection Lookaside Buffer
CPU	Central Processing Unit
CMOS	Complementary Metal Oxide Semiconductor
DACR	Domain Access Control Register
DICE	Dual Interlocked Storage Cell
ECC	Error-Correcting Code
ECU	Electronic Control Unit
EIS	Book E Implementation Standards
FH	Fault Handler
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HS	Hit Selector
HTW	Hardware Table Walk
IC	Integrated Circuit
IO	Input/Output
IP	Intellectual Property
IPA	Intermediate Physical Address
LE	Logic Element
LPAE	Large Physical Address Extension
LRU	Least Recently Used
PC	Personal Computer
PID	Process Identifier
PXN	Privileged Execute Never
Tcl	Tool Command Language
TLB	Translation Lookaside Buffer
MAS	MMU Assist
MEM	Memory

MI	Memory Interface
MMU	Memory Management Unit
MPU	Memory Protection Unit
OS	Operating System
PAE	Physical Address Extension
PTE	Page Table Entry
PTD	Page Table Descriptor
RAM	Random-Access Memory
REQ	Request
RES	Response
RTL	Register-Transfer Level
SEC-DED	Single-Error Correction and Double-Error Detection
SEE	Single Event Effect
SEFI	Single Event Functional Interrupt
SEL	Single Event Latch-up
SER	Soft Error Rate
SET	Single Event Transient
SEU	Single Event Upset
SMU	Single-word Multiple-bit Upset
SPR	Special-Purpose Register
SRAM	Static RAM
TM	TLB Manager
TMR	Triple Modular Redundancy
TRS	TLB Replacement Strategy
TSE	TLB Storage Element
TID	Translation ID
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VMID	Virtual Machine Identifier
VPNE	Virtual Page Number Extraction
XN	Execute-never

Bibliography

[Aer12] AEROFLEX GAISLER: *GRLIB IP Core User's Manual*. Version 1. 2012 (January). http://www.gaisler.com/products/grlib/grlib.pdf

[Alt11] ALTERA CORPORATION: *Nios II Processor Reference Handbook*. 11. 2011 (May). http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf

[AM73] ANDERSON, Douglas A. ; METZE, Gernot: Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes. In: *IEEE TRANSACTIONS ON COMPUTERS* C-22 (1973), Nr. 3, 263–269. http://dx.doi.org/10.1109/T-C.1973.223705. – DOI 10.1109/T–C.1973.223705

[AMD12] AMD: *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. 2012 http://support.amd.com/us/Embedded_TechDocs/24593.pdf

[Ana10] ANALOG DEVICES INC.: *ADSP-BF54x Blackfin Processor Hardware Reference*. 2010 http://www.analog.com/static/imported-files/processor_manuals/ADSP-BF54x_HRM_rev1.0.pdf

[ARM11] ARM LIMITED: *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. 2011 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html

[ARM12a] ARM LIMITED: *Cortex-A15 MPCore Technical Reference Manual*. r3p0. 2012 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438e/index.html

[ARM12b] ARM LIMITED: *Cortex-A9 Technical Reference Manual*. 2012 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388h/index.html

[Bau05] BAUMANN, R.C.: Radiation-induced soft errors in advanced semiconductor technologies. In: *IEEE Transactions on Device and Materials Reliability* 5 (2005), September, Nr. 3, 305–316. http://dx.doi.org/10.1109/TDMR.2005.853449. – DOI 10.1109/TDMR.2005.853449. – ISSN 1530–4388

[CG05] CHOUDHURI, Siddharth ; GIVARGIS, Tony: Software Virtual Memory Management for MMU-less Embedded Systems / Center for Embedded Computer Systems, University of California. Version: 2005. http://www.cecs.uci.edu/technical_report/TR05-16.pdf. Irvine, 2005. – Forschungsbericht

[CNV96] CALIN, T. ; NICOLAIDISL, M. ; VELAZCO, R.: Upset Hardened Memory Design for Submicron CMOS Technology. In: *IEEE Transactions on Nuclear Science* 43 (1996), Nr. 6, 2874–2878. http://dx.doi.org/10.1109/23.556880. – DOI 10.1109/23.556880

[Con02] CONSTANTINESCU, C.: Impact of deep submicron technology on dependability of VLSI circuits. In: *Proceedings International Conference on Dependable Systems and Networks* (2002), 205–209. http://dx.doi.org/10.1109/DSN.2002.1028901. – DOI 10.1109/DSN.2002.1028901. ISBN 0–7695–1597–5

[Eis02] EISELE, Konrad: *Design of a Memory Management Unit for System-on-a-Chip Platform "LEON"*, University of Stuttgart, Dipoma Thesis, 2002

[EJ09] EBERT, Christof ; JONES, Capers: Embedded Software: Facts, Figures, and Future. In: *Computer* 42 (2009), April, Nr. 4, 42–52. http://dx.doi.org/10.1109/MC.2009.118. – DOI 10.1109/MC.2009.118. – ISSN 0018–9162

[EME94] ELGUIBALY, Hani ; MUZIO, Jon ; ELGUIBALY, Fayez: On concurrent error detection of finite state machine systems. In: *Proceedings of the 37th Midwest Symposium on Circuits and Systems*, 1994, 213–216

[Fre05] FREESCALE SEMICONDUCTOR INC.: *Freescale PowerPC Architecture Primer*. Rev. 0.1. 2005 http://www.freescale.com/files/32bit/doc/white_paper/POWRPCARCPRMRM.pdf

[Fre09] FREESCALE SEMICONDUCTOR INC.: *e200z4 Power Architecture™ Core Reference Manual*. 2009 http://cache.freescale.com/files/32bit/doc/ref_manual/e200z4RM.pdf

[GAN+10] GADLAGE, Matthew J. ; AHLBIN, Jonathan R. ; NARASIMHAM, Balaji ; BHUVA, Bharat L. ; MASSENGILL, Lloyd W. ; REED, Robert A. ; SCHRIMPF, Ronald D. ; VIZKELETHY, Gyorgy: Scaling Trends in SET Pulse Widths in Sub-100 nm Bulk CMOS Processes. In: *IEEE Transactions on Nuclear Science* 57 (2010), Nr. 6, S. 3336–3341

[GMB+99] GIL, D ; MARTINEZ, R ; BUSQUETS, J. V. ; BARAZA, J. C. ; GIL, P. J.: Fault Injection into VHDL Models: Experimental Validation of a Fault-Tolerant Microcomputer System. In: *Dependable Computing—EDCC-3* (1999), 191–208. http://www.springerlink.com/index/c3d09up834v1vqbv.pdf

[GNP05] GILL, B. ; NICOLAIDIS, M. ; PAPACHRISTOU, C.: Radiation induced single-word multiple-bit upsets correction in SRAM. In: *11th IEEE International On-Line Testing Symposium* (2005), 266–271. http://dx.doi.org/10.1109/IOLTS.2005.59. – DOI 10.1109/IOLTS.2005.59. ISBN 0–7695–2406–0

[HGS05] HUNG, L.D. ; GOSHIMA, M. ; SAKAI, S.: Mitigating soft errors in highly associative cache with CAM-based tag. In: *2005 International Conference on Computer Design* (2005), 342–347. http://dx.doi.org/10.1109/ICCD.2005.76. – DOI 10.1109/ICCD.2005.76. ISBN 0–7695–2451–6

[HMHO12] HARADA, Ryo ; MITSUYAMA, Yukio ; HASHIMOTO, Masanori ; ONOYE, Takao: SE-1 SET Pulse-Width Measurement Eliminating Pulse-Width Modulation and Within-Die Process Variation Effects. In: *Reliability Physics Symposium (IRPS), 2012 IEEE International*, 2012. – ISBN 9781457716805, SE.1.1–SE.1.6

[HP09] HENNESSY, John L. ; PATTERSON, David A.: *Computer Organization and Design: The Hardware/Software Interface*. 4. Morgan Kaufmann Publishers, 2009. – ISBN 9780123744937

[HP12] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 2012. – ISBN 9780123838728

[JAR+94] JENN, E. ; ARLAT, J. ; RIMEN, M. ; OHLSSON, J. ; KARLSSON, J.: Fault injection into VHDL models: the MEFISTO tool. In: *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing* (1994), 66–75. http://dx.doi.org/10.1109/FTCS.1994.315656. – DOI 10.1109/FTCS.1994.315656. ISBN 0–8186–5520–8

[jes06] *JEDEC STANDARD JESD89A: Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, 2006 (OCTOBER)

[LKJ+12] LOVELESS, T. D. ; KAUPPILA, J. S. ; JAGANNATHAN, S. ; BALL, D. R. ; ROWE, J. D. ; GASPARD, N. J. ; ATKINSON, N. M. ; BLAINE, R. W. ; REECE, T. R. ; AHLBIN, J. R. ; HAEFFNER, T. D. ; ALLES, M. L. ; HOLMAN, W. T. ; BHUVA, B. L. ; MASSENGILL, L. W.: On-Chip Measurement of Single-Event Transients in a 45 nm Silicon-on-Insulator Technology. In: *IEEE Transactions on Nuclear Science* 59 (2012), Dezember, Nr. 6, 2748–2755. http://dx.doi.org/10.1109/TNS.2012.2218257. – DOI 10.1109/TNS.2012.2218257. – ISSN 0018–9499

[Nic05] NICOLAIDIS, M.: Design for soft error mitigation. In: *IEEE Transactions on Device and Materials Reliability* 5 (2005), September, Nr. 3, 405–418. http://dx.doi.org/10.1109/TDMR.2005.855790. – DOI 10.1109/TDMR.2005.855790. – ISSN 1530–4388

[SPA92] SPARC INTERNATIONAL INC.: *The SPARC Architecture Manual Version 8*. Bd. 8. 8. 1992 http://www.sparc.org/standards/V8.pdf

[Tan09] TANENBAUM, Andrew S.: *Modern Operating Systems*. 3. Pearson Education, Inc., 2009. – ISBN 9780138134594

[WW05] WADDLE, J. ; WAGNER, D.: Fault Attacks on Dual-Rail Encoded Systems. In: *21st Annual Computer Security Applications Conference (ACSAC 2005)* (2005), 483–494. http://dx.doi.org/10.1109/CSAC.2005.25. – DOI 10.1109/CSAC.2005.25. ISBN 0–7695–2461–3

[Xil12] XILINX: *TMRTool Product Brief*. 2012 http://www.xilinx.com/publications/prod_mktg/CS11XX_TRMTool_Product%_Brief_FINAL.pdf

[ZAV04] ZIADE, Haissam ; AYOUBI, Rafic ; VELAZCO, Raoul: A Survey on Fault Injection Techniques. In: *The International Arab Journal of Information Technology* 1 (2004), Nr. 2, S. 171–186