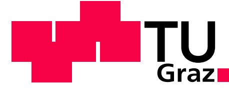

Michael Stefan Maierhofer, BSc

Automated generation of a Module Protection Unit for RISC-V based SoCs following the Model-driven Architecture principle

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inf. Dr.rer.nat. Marcel Carsten Baunach
Institute of Technical Informatics

Dipl.-Ing. Heimo Hartlieb
Infineon Technologies Austria AG

Graz, October 2020

This document is set in Palatino, compiled with pdfLaTeX 2_ε and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

The number of transistors in ICs is still steadily growing despite Moore's law being declared dying numerous times. The productivity is trailing behind with an increase by factor 20, leading to an ever-expanding 'design-productivity gap'. The closing of this gap has led to the introduction of a novel hardware generation flow based on a model-driven architecture (MDA) principle.
Prior to this thesis, a CPU core generator for the hardware MDA flow had been developed, based on the RISC-V ISA. Said core had support for interrupts and exceptions but was missing means to restrict accesses to memory and peripherals. Such restrictions are crucial to ensure freedom from interference between subsystems in functional safety systems. This work looks into the generation of a memory protection unit (MPU) as an extension to the MDA flow for RISC-V based SoCs.
The MPU presented in this work is fully incorporated into the core and enables effective memory protection and isolation with a minimal performance overhead.

Kurzfassung

Die Anzahl an Transistoren in ICs steigt trotz dem mehrfach erklärten Ende des Mooreschen Gesetz stetig weiter. Gleichzeitig fällt die Produktivität immer weiter zurück. Bereits jetzt ist das Produktivitätsdefizit um mehr als das 20-fache gestiegen, und es wird immer größer. Um diesen „Design-Productivity Gap“ zu schließen wurde ein neuartiger Ansatz für die Generierung von Hardware basierend auf dem Prinzip einer Modell-getriebenen Architektur (MDA) entwickelt.
Vor dieser Masterarbeit wurde bereits ein Generator für eine CPU mit dem RISC-V Instruktionssatz für den Hardware MDA-Flow entwickelt. Diese CPU unterstützte Interrupts und Exceptions, eine Möglichkeit Zugriffe auf Speicher sowie Peripherie zu beschränken waren allerdings nicht gegeben. Derartige Zugriffsbeschränkungen sind allerdings notwendig um die Rückwirkungsfreiheit (engl. „freedom from interference“) zwischen Subsystemen in Systemen mit Anforderungen an Funktionaler Sicherheit zu garantieren. Weswegen sich diese Arbeit nun damit beschäftigt eine Memory Protection Unit (MPU) als Erweiterung des MDA-Flow für RISC-V basierte SoCs zu realisieren.
Die im Zuge dieser Arbeit entworfene MPU ist vollständig in die CPU integriert, und ermöglicht einen effektiven Speicherschutz und Isolation mit minimalen Leistungseinbußen.

Acknowledgements

I want to thank both Univ.-Prof. Dipl.-Inf. Dr.rer.nat. Marcel Carsten Baunach and Dipl.-Ing. Heimo Hartlieb for the supervision and the opportunity to write this Master's Thesis in cooperation with Infineon Technologies AG. Thank you for sharing your expertise and experience in the field as well as the guidance during the creation of this work.

I also want to thank all my friends and colleagues I met during my studies or that accompanied me during this time. Studying would not have been the same without you and your support.
Furthermore, I especially want to thank Paul and Dominic. You helped me stay positive and sane over the emotional rollercoaster that was finding and creating a Master's Thesis.

Last but not least, I want to thank my family and parents. Thank you for your continuous support and for always letting me pursue my dreams.

Contents

1. Introduction **1**
1.1. Background 1
1.2. Objective and Motivation 2
1.3. Related Work 4
1.4. Outline 6

2. Model Driven Architecture **7**
2.1. Model Driven Architecture Concept 7
2.2. Metagen and MetaRTL 11

3. Hard- and Software Assumptions and Constraints **13**
3.1. Software Assumptions 13
3.2. Hardware Assumptions and Constraints 14
3.3. Memory Partitioning and MPU Registers 15

4. Extension to the CSR model **22**
4.1. CSR Template and Model of Design 23
4.2. MPU Extension 29

5. MDA based MPU generation **33**
5.1. MPU Placement in the RISC-V Model of Things 33
5.2. MPU Template and Model of Design 35
5.3. MPU Placement within the CPU Pipeline 48

6. Evaluation and Results **50**
6.1. Behavioural Tests 50
6.2. Resource Requirements 55
6.3. MPU Application Evaluation 59
6.3.1. Application Architecture 59
6.3.2. Application Example 61

6.3.3. Results 64

7. Summary and Future Work **68**

Bibliography **70**

A. Behavioural Test Source Code **75**
A.1. Kernel and Exception Handler 75
A.2. Exception Handling Test 78
A.3. Configuration Change Test 80

B. Application Performance Test Source Code **83**
B.1. Dhrystone Source Code 83
B.2. Kernel and Exception Handler 100
B.3. MPU Management 104

List of Figures

1.1. SoC architecture overview of the 'RiVal' test chip 3

2.1. MDA as Y-Chart 8
2.2. MDA for hardware generation 10
2.3. Metamodel of Metapad 11

3.1. MPU configuration CSR layout 16
3.2. MPU address register format 17
3.3. MPU configuration entry (cfg) layout 17
3.4. MPU control register format 21
3.5. MPU status CSR layout 21

4.1. Extended UML diagram of the CSR classes 24
4.2. CSR RegisterFile Structure 26
4.3. CSR Structure with no Bitfields 27
4.4. CSR Structure with Bitfields 28
4.5. Address Register Structure 30
4.6. Configuration Register Bitfield Structure 31

5.1. Schematic metamodel of RISC-V core (MetaRISC) expanded with MPU 34
5.2. UML diagram of the MPU classes 35
5.3. MPU toplevel Structure 38
5.4. MPU Status Structure 39
5.5. MPU Decoder Structure 40
5.6. MPU TOR Structure 41
5.7. MPU NAPOT Structure 42
5.8. MPU Access Check Structure 44
5.9. MPU Execute Access Structure 45
5.10. MPU Read/Write Access Structure 46

5.11. MPU Fault Logic Structure 47

6.1. Decoder test 50
6.2. Waveform of the exception handling test 52
6.3. Waveform of the configuration change test 54
6.4. Dhrystone Benchmark runtimes 66

List of Tables

3.1. 'A' field encoding of configuration registers of MPU 18
3.2. NAPOT range encoding in address and configuration registers of MPU 19
3.3. 'B' field encoding of configuration registers of MPU 20

6.1. Timing report of MPU 55
6.2. Cell report of MPU 57
6.3. Area report of MPU 57
6.4. Cell report of 'RiVal' w/wo MPU 58
6.5. Area report of 'RiVal' w/wo MPU 58
6.6. Dhrystone Benchmark Results 64

List of Listings

2.1. MetaRTL example of a half adder structure 12

6.1. MPU region structure 59
6.2. MPU configuration structure 60
6.3. MPU configuration load function 62
6.4. MPU address write function 63

A.1. kernel.c for behavioural tests 76
A.2. kernel.S 78
A.3. exception_test.c 80
A.4. change_test.c 82

B.1. dhry_1.c 94
B.2. dhry_2.c 100
B.3. kernel.c for Dhrystone 101
B.4. kernel.S for Dhrystone 103
B.5. mpu.h 104
B.6. mpu.c 111

Terms and Abbreviations

AHB	Advanced High-performance Bus
API	application programming interface
ASIC	application specific integrated circuit
ASIL	automotive safety integrity level
AST	abstract syntax tree
CIM	computation independent model
CISC	complex instruction set computer
CPU	central processing unit
CSR	control status register
ECU	electronic control unit
FPGA	field programmable gate array
HDL	hardware description language
HGL	hardware generation language
HW	hardware
IC	integrated circuit
IDE	integrated development environment
IO	input/output
IoT	Internet-of-Things
IP	Intellectual Property
ISA	instruction set architecture
ISO	International Organization for Standardization
LSB	least significant bit
LuT	lookup table
MDA	model-driven architecture
MMU	memory management unit
MoC	Model of Computation
MoD	Model of Design
MoT	Model of Things
MoV	Model of View

MPU memory protection unit
MSB most significant bit
NoC network-on-chip
OMG Object Management Group
OS operating system
PC program counter
PIM platform independent model
PM platform model
PMA physical memory attribute
PMP physical memory protection
PSM platform specific model
PTS powertrain and safety
RAM random access memory
RISC reduced instruction set computer
ROM read only memory
RTL register transfer level
RTOS real time operating system
SoC system-on-chip
SW software
TLB translation lookaside buffer
ToD Template of Design
ULP ultra-low power
UML unified modelling language
UUT unit under test
XML extended modelling language

1. Introduction

1.1. Background

The number of transistors in integrated circuit (IC) is still steadily growing despite Moore's law being declared dying numerous times. According to a study by McKinsey in 2013, the number of transistors that can be manufactured increased by factor 100 per decade, while productivity is trailing behind with an increase by factor 20, leading to an ever-expanding 'design-productivity gap' [CP13]. Closing this gap is essential for all leading companies in the industry.

An approach commonly used in practice is 'IP-reuse'. Rather than a (re-)development from scratch, pre-existing blocks or whole systems are adapted or used without further modifications in the new system. Typical examples are RAM or ROM modules. This leads to an increase in productivity as well as possible lower development costs.
The evolution to the aforementioned method is complete system or code generation from abstract descriptions. This hardware generation approach was named as the next disruptive productivity improvement after IP-reuse by leading research groups in the industry [ES16]. This code and system generation approach is continuously evolving. At the same time, it was shown that a 20x increase in productivity in particular design tasks and 3x higher productivity during implementation from specification freeze to tape-out can be achieved [Eck+14]. A well-known example of the usage of code generation was the development of a hardware generation language (HGL) called Chisel by Bachrach et al. [Bac+12]. The approach taken by Chisel demonstrated a 10-fold code reduction compared to Verilog code.

This development has led to the creation of Infineon's hardware generation flow, replacing the manual hardware description language (HDL) generation

with a high-level generator [Zap18] (see Chapter 2 for details).
With Infineon's approach to hardware generation, a wide variety of components and peripherals (e.g., RAM, ROM, interrupt controller, timer, etc.) have been implemented. The newest addition to this list is a fully verified RISC-V core [Sch16], enabling the creation of a fully customisable embedded system-on-chip (SoC). Logically, the next step for Infineon as a company is to evaluate if it is possible and feasible to develop chips using only its in-house generation methodology and have them manufactured in silicone.
This has lead to the creation of a test chip (i.e. a ISO 26262 compliant (as defined by the International Organization for Standardization (ISO) [ISO18]) SoC for the powertrain and safety (PTS) automotive market) featuring an open-source RISC-V ISA core called 'RiVal' (RISC-V test chip for ASIL compliant applications).

A study conducted prior to the creation of this thesis revealed that in order to allow for functional safety compliant software development the test chip is missing means to restrict accesses to memory and peripherals.

1.2. Objective and Motivation

Functional safety certification standards such as IEC 61508 and domain-specific derivatives such as ISO 26262 require that a failure in one application cannot influence the behaviour of another. This freedom of influence is especially important for safety-critical systems. Such systems often use shared-memory architectures, while convenient to use, resource sharing constitutes a problem when separation is required. For example, in such architectures without protection, each application can access and manipulate the data of other applications [HRK12].

To prevent processes or applications from interfering with each other, a common approach is to have a memory protection unit (MPU) or memory management unit (MMU) included in the central processing unit (CPU). MPUs are used to restrict access to predefined memory regions, which are usually configured by an operating system (OS). This region partitioning also partitions the software stack (e.g., in safety-related software and non-safety-related software) as required by functional safety standards [YN14]. The MPU then

verifies that the currently running process only accesses the address space it is eligible to access. Upon the context switch to another process, the OS updates the MPU configuration for the new process. Instead of the necessary configuration changes when using MPUs, MMUs can be used. In MMU based systems each process has its own virtual address space. When a process tries to access a virtual address, the MMU looks it up in the page table and translates it to a physical address. During context switches the OS only needs to write the page address of the next process to the MMU [HRK12].

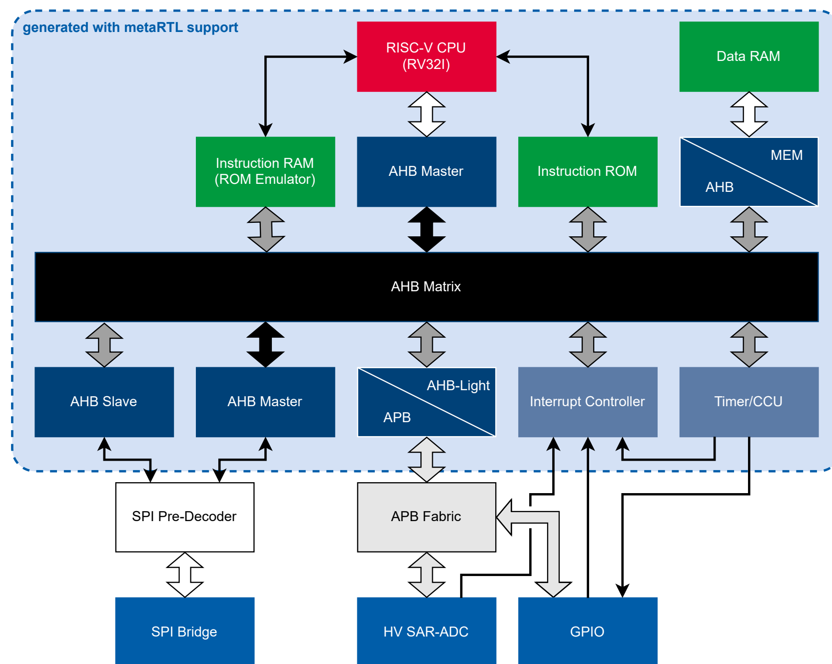

Figure 1.1.: SoC architecture overview of the 'RiVal' test chip

To allow process separation on the 'RiVal' platform, the objective of this thesis is to create a memory protection module (either an MMU or MPU) with the help of Infineon's hardware generation flow. A depiction of this platform can be seen in Figure 1.1. Each of the blocks depicted in this figure was created with metaRTL support (details regarding metaRTL can be found in Section 2.2). These blocks are called modules in Infineon's terminology. Therefore, the

proposed unit in this thesis to protect these modules from illegal accesses is named module protection unit (MPU). In the context of this thesis, the terms memory protection unit and module protection unit are both abbreviated with MPU and used interchangeably.

MMUs offer greater flexibility and more access control features compared to MPUs. This increase in performance and flexibility, however, comes at the cost of an increase in area size. They require for their address translation a translation lookaside buffer (TLB) to cache table entries as well as MMU translation tables. To keep the increase in area size and therefore the cost of the future SoC at a minimum it was consequently decided to go with an MPU instead.

1.3. Related Work

Various authors leverage pre-existing protection units such as the MPU contained in ARM Cortex-M and Cortex-A [19a], or Infineon AURIX [19b] processors, to isolate and/or protect memory.
Pan and Parmer [PP19] propose an MxU to provide memory protection and allocation abstraction. This MxU enables both tightly-bounded execution and dynamic memory management of portable code on Internet-of-Things (IoT) devices. Lopriore [Lop16] presented a model of a protection system based on passwords for embedded systems. To reduce the overhead, the model leverages an MPU interposed between the processor and memory. The proposed system's focus is on system security and less on system safety. The authors of [FDM19] and [YN14] focus on memory protection mechanisms in real time operating systems (RTOSs) using the MMU in ARM processors. The presented approaches try to minimise the overhead imposed by context switches between tasks and to provide a hardware abstraction interface to the operating system. Rivera [Riv18] describes how an MPU can be used to provide runtime data protection and isolation within a single address space of an embedded system using bare-metal software written in Ada.
IO and peripheral virtualisation is another approach commonly used to encapsulate each device into its own address space. This virtualisation is used by the operating system to protect itself against buggy drivers or malicious devices. Malka et al. [Mal+15] showed an approach aimed at high-performance

IO devices such as 1000 Gbps network controllers. The described IOMMU only makes sense to be included in desktop computers or powerful ECUs since low-power embedded systems usually do not have such powerful peripheral devices. A lightweight IO virtualisation which leverages the MPU of ARM Cortex-M MCUs was described by Paci, Brunelli and Benini [PBB18]. Their virtualisation layer integrates with FreeRTOS and supports dynamic linking of new user code.

Ultra-low power (ULP) processors have special requirements when it comes to memory management and protection. Such ULP systems (e.g., sensor node networks) are often distributed heterogeneous multiprocessor systems with shared memory. The stringent low-power constraint often means that the processor design cannot be changed. For such networks an MMU or MPU can be used that is embedded in a network-on-chip (NoC). Multiple systems which use this approach have been proposed, most notably by Jang et al. [Jan+19], Porquet, Greiner and Schwarz [PGS11] and Hattendorf, Raabe and Knoll [HRK12]. For other devices that do not share memory (e.g., IoT devices, microcontrollers, etc.) a more classical approach of integrating an MPU or MMU into the processor can be taken. Stecklina, Langendoerfer and Menzel [SLM13] propose an MPU design for a low power 16-bit microcontroller (IHP430x) with only a 7% increase in total chip area. The small size, however, comes at the cost that it only supports up to 16 protection regions with limited flexibility in terms of usage of those regions and no execution protection. A similar approach was also taken by Lopriore [Lop14]. Shamani et al. [Sha+16] described a 64 region MMU that was integrated into the COFFEE core (i.e., a RISC processor developed by a group at Tampere University of Technology, Finland).

MMUs and MPUs have been of interest for both academia and industry for many years. As expected, a lot of architectures and implementations were proposed for different use cases and applications. Nevertheless, no publication was found on model-driven architecture (MDA) driven automated MPU generation during the creation of this work. Therefore, the proposed approach of this thesis is a novelty.

1.4. Outline

The remainder of this work is structured as follows: Chapter 2 describes the principles of model-driven architecture (MDA) and the metamodelling approach for hardware generation. In Chapter 3, the set constraints and assumptions made to the framework, hardware platform, and expected software usage are looked into.
The required changes to models, characteristics, and detailed descriptions of the implementations are discussed in Chapters 4 and 5. The former covers the extension to the control status register (CSR) model for the configuring of the new module, while the latter addresses the generation of the MPU.
The tests and evaluations of the implementation are discussed in Chapter 6.
And finally, current limitations and possible enhancements as future work are looked into in Chapter 7.

2. Model Driven Architecture

2.1. Model Driven Architecture Concept

Model-driven architecture (MDA) is a vision coined by the Object Management Group (OMG), an international industrial consortium that creates and maintains specifications for software interoperability [MCF03]. The OMG also addresses future software design aiming to reduce the growing productivity gap (i.e., the available workforce versus the required workforce). In an MDA development flow, the primary artefacts throughout the development cycle are models [KM05]. By capturing these artefacts in a formalised way, automated processing, starting from very abstract specification all the way to the actual implementation, is enabled. This formalisation is called Metamodelling and is, therefore, an integral part of MDA. Doing so a way is provided to automatically transform and refine models into more specific and fine-grained models. Eventually, the most granular model is used to generate the intended target.

Figure 2.1 shows the three abstraction layers originally introduced in MDA, as well as a forth, PM, containing details of the target platform. These layers are defined in such a way that the target code is generated starting from a high-level human-elaborated definition (diagrams, models, etc.). The models on each of these layers are formalised by a metamodel they are an instance of. It should be noted that the metamodel is a model as well and is formalised by a so-called meta-metamodel. Metametamodels are used, for example, to generate metamodels. It is used to generate parts of the metamodelling approach applied to this thesis. Discussion and further details go beyond the scope of this work, as such, the interested reader is referred to [ES16].
The aforementioned layers are:

- *Computation independent model (CIM)* is very abstract and close to the

specification. Neither details about algorithm implementation nor architecture are considered.

- *Platform independent model (PIM)* avoids platform details, but already defines the architecture and therefore sets constraints for the implementation at a high level of abstraction.
- *Platform model (PM)* contains details of the target platform and the technology being used.
- *Platform specific model (PSM)* adds the utilised platform to the PIM and therefore, is platform-dependent and closest to the target code (also referred to as view). From this model, the view (i.e., e.g. target code) is generated.

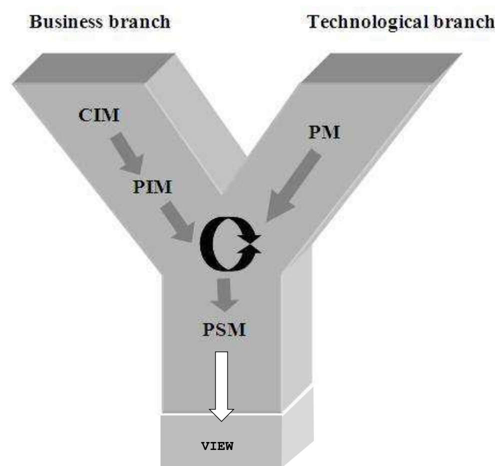

Figure 2.1.: MDA as Y-Chart (reprinted from [ES16])

The vanilla MDA approach itself does not map that well to hardware design, which is why an adaptation for digital hardware generation was necessary. The adoption done by Ecker and Schreiner [ES16] follows the conceptual three-layer model of the already described MDA but enhances it to support hardware design. This is achieved by introducing new terms that describe the involved hardware related models, as seen in Figure 2.2. Again, every model on each layer is formalised by its corresponding metamodel.

- *Model of Things (MoT)* corresponds to the level of the CIM as it is intended to formally capture requirements and specifications. It describes

the functionality an implementation provides while leaving out implementation details such as system architecture. One example of an MoT is an instruction set architecture (ISA) of a CPU.

- *Model of Design (MoD)* corresponds to the level of the PIM and could be described as the core model of this methodology. Its goal is to define the architecture using the designer's terms. It models the intended functionality in the MoT. It should be noted that the MoD does not include information on how individual components are provided on a particular target platform. Doing so avoids the introduction of artefacts from simulation or synthesis. For example, the MoD of a memory subsystem will describe its components and characteristics of read and write ports; it will not, however, describe how those are implemented [Sch16].
- *Model of View (MoV)* corresponds to the level of the PSM since it is the least abstract model. It has a straight forward mapping to the target view. Implementation details such as target hardware dependencies or technologies from FPGAs or ASICs are added during the transformation from MoD to MoV. The MoV corresponds to a language-specific abstract syntax tree (AST), from which RTL files can be generated. Furthermore, it also implicitly determines the Model of Computation (MoC) as this is inherently defined by the target view.

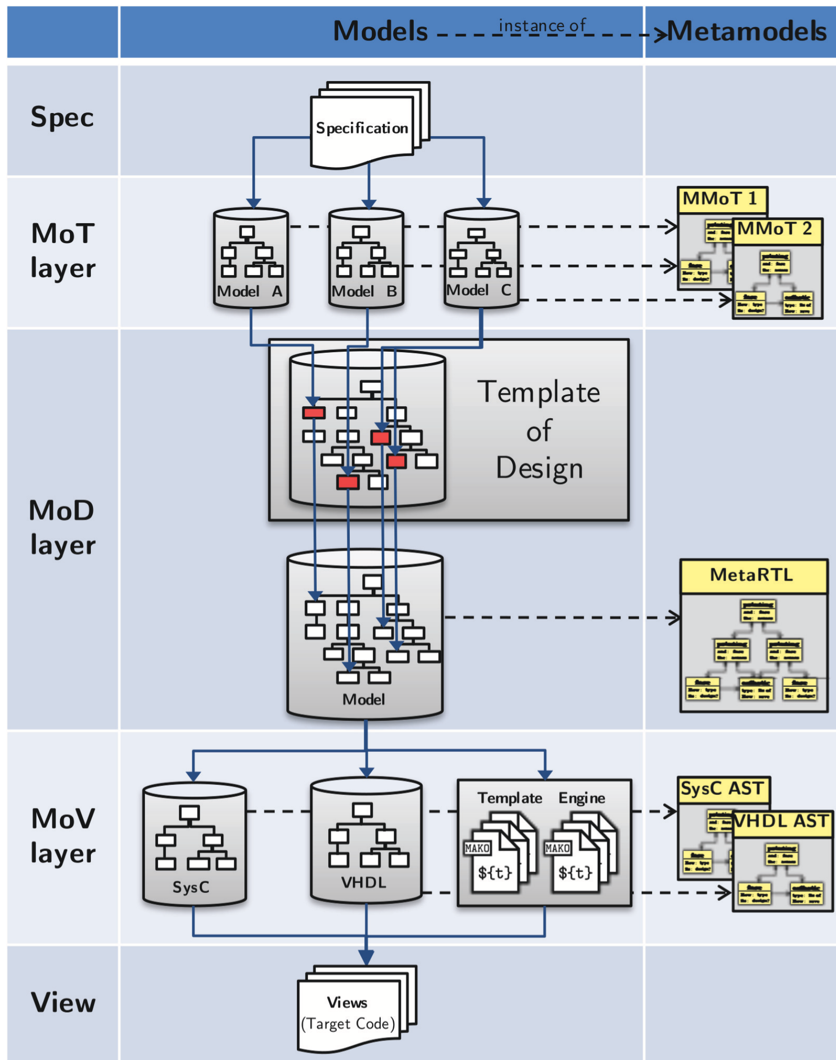

Figure 2.2.: MDA for hardware generation (reprinted from [Sch16])

2.2. Metagen and MetaRTL

Metagen is Infineon's framework, implementing the three modelling layers of Metamodelling and code generation described in Section 2.1. The framework uses a UML subset to specify metamodels. These class diagrams are extended utilising XML-like specifications of alternatives, subsequently allowing for the specification of objects, attributes, and their relations between them. Meta-metamodels are used to relate and combine known metamodels. They are also used to generate metamodels from other formalisms such as XML schemata.

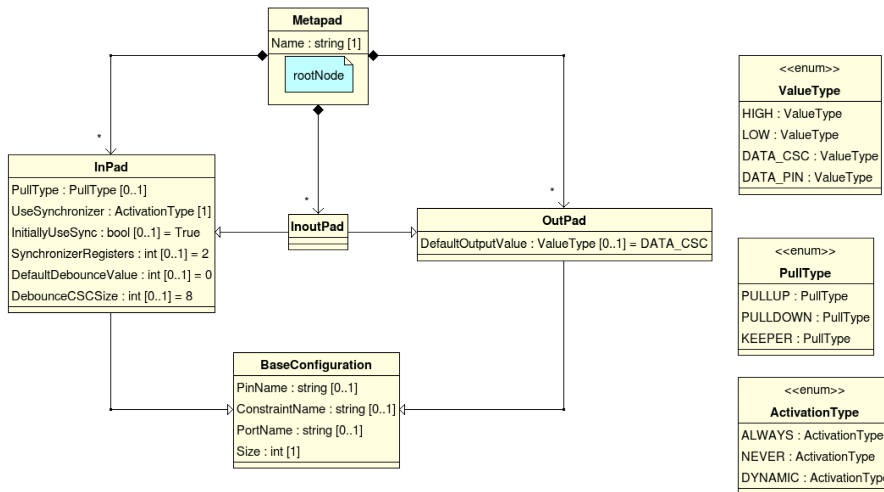

Figure 2.3.: Metamodel of Metapad (reprinted from [Pau18])

Metamodels can be captured both graphically and textual. The infrastructure generated by the Metagen framework supports the generation of extensible application programming interfaces (APIs), code generation for persistent storage, etc. As the framework is written entirely in Python, it takes advantage of numerous Python libraries and tools, including the Mako template engine for MoV generation [ES16]. An example MoT metamodel can be seen in Figure 2.3. This model shows the UML representation of Metapad (i.e., a model representing a pad).

MetaRTL is the metamodel developed to formalise the model of the MoD layer.

It provides a set of components (e.g., multiplexer, register, etc.), as well as methods of various types needed to generate RTL structures (the output can be either VHDL or Verilog), thus building a Template of Design (ToD). This ToD is part of MoD layer (as seen in Figure 2.2) consisting of a set of constructors for the MoD, where each constructor instantiates a specific component depending on the data and information contained in the MoT. It describes a blueprint on how the MoD can be built.

Infineon's implementation of the ToD is pure Python code, using a set of predefined packages and classes. Python environments give the ability to use numerous libraries, modern programming concepts, and to utilise Python IDEs. An example of a half adder structure written as part of a ToD can be seen in Listing 2.1.

```
1 class halfAdder(Structure):
2   def __init__(self, parent=None):
3     super(halfAdder, self).__init__(
4       Ports=[{'Name': 'in0', 'Direction': 'IN'},
5              {'Name': 'in1', 'Direction': 'IN'},
6              {'Name': 'sum', 'Direction': 'OUT'},
7              {'Name': 'carry', 'Direction': 'OUT'}], parent=parent)
8       self['sum'].connect(LXOR(self['in0'], self['in1']))
9       self['carry'].connect(LAND(self['in0'], self['in1']))
```

Listing 2.1.: MetaRTL example of a half adder structure

3. Hard- and Software Assumptions and Constraints

Before any implementation can be written, constraints and assumptions to hardware as well as software need to be examined. These constraints and assumptions are analysed in this chapter.

3.1. Software Assumptions

The first assumption in regards to software is that all code is generally trusted, immutable, and not self-modifying. This also implies that there will be no possibility of binary changes during runtime. Those assumptions are easily justified when writing software in accordance with functional safety standard ISO 26262, as those requirements ease certification [ISO18]. The trusted code requirement can be achieved through code review, which is also commonly used as a best practice in software development.

The next assumption regards the platform (i.e., the hardware that software is run on) the software is created for. It is assumed that the platform is a bare-metal embedded system. Effectively meaning that the software stack will be as slim as possible with no, or minimal, support of an operating system (OS) or real time operating system (RTOS). This assumption is made as during the creation of this thesis the SoCs developed at the respective Infineon department were mainly used as a hardware controller with a communication interface to a more powerful (automotive) electronic control unit (ECU).

Often, controllers in such systems only support a single privilege or execution mode, meaning that all software is run at the same (highest) privilege level. For security and safety reasons, however, it is assumed that multiple privilege

levels are available, e.g., machine and user mode. The official RISC-V definition can be found in [WA19a] and [WA19b]. Having multiple modes available allows the software flow to be split into multiple tasks or modes (e.g., boot, test, mission mode, etc.) all running at the same low privilege level (i.e., user-level), with only one task as central trusted entity running at a higher privilege level (i.e., machine level). Such a trusted entity could, for example, be the kernel of an (RT)OS. The kernel task's responsibility is to load the corresponding MPU configuration of a task during the task dispatching. These configurations are stored in a read-only memory section (e.g., read only memory (ROM)), thus being not modifiable. The other tasks (i.e., user-mode tasks) do not modify the currently loaded MPU themselves. This behaviour can be further enforced using hardware locking the currently loaded configuration. Once a configuration is locked in place only a task running at a higher privilege level (i.e., machine mode task) or a system reboot can unlock it. See Section 3.3 for details.

3.2. Hardware Assumptions and Constraints

In regards to hardware constraints, the most important one is that the MPU will be integrated seamlessly into the MetaRISC metamodel (as seen in Figure 5.1) and will be generated on demand with Infineon's MDA flow described in Chapter 2. This results in several implications. Most importantly, the MPU will be part of the CPU, and not connected as an external module via a bus interface such as Advanced High-performance Bus (AHB). The CPU design during the time of writing this thesis is a 5-stage pipelined CPU design that only supports the RV32I instruction set (version 2.1). Therefore, the CPU pipeline needs to be considered for the MPU placement. Furthermore, an exception unit has been built for the used RISC-V core by Zappia [Zap18], and thus the MPU should work with this exception unit. Finally, the number of possible MPU registers is constrained by the fact that they have to be mapped into the custom control status register (CSR) address space according to the RISC-V ISA (see [WA19b] for details).

The next assumptions effect the number of potential protection regions. Given the dynamic potential of the MDA flow, it is assumed that it would be beneficial to have a flexible number of possible MPU protection regions. Just as many as required for a specific application. When choosing the number of required

regions it should be kept in mind that if an access control granularity of 1 byte is necessary, each 1 byte region needs its MPU region. The modules to and from which the access shall be controlled are assumed to be memory-mapped into the databus address space.

The MPU usage and specification should follow the physical memory protection (PMP) and physical memory attribute (PMA) specifications from the privileged RISC-V ISA specification (Version 1.11) (see [WA19b] for details) as closely as possible. Therefore, allowing for Intellectual Property (IP) reuse in case an MMU in accordance to the RISC-V ISA is needed in future. The MMU described by the RISC-V specification is powerful indeed. However, its complete implementation would result in an excessive overhead in terms of features and more important area size. This is because the implementation or usage of a single PMP or PMA register requires the implementation of all other registers as well. The inclusion of these registers is not generally necessary, but rather it is an explicit requirement from the RISC-V specification. The MMU described in the RISC-V specification also supports advanced features such as virtual memory, which is something that is hardly ever seen and used in bare-metal embedded systems and as such would waste precious chip area.

Lastly, at the time of writing this thesis, no other privilege modes other than machine mode of the core was available. The lack of privilege modes directly contradicts the software assumptions made in Section 3.1, leading to a restriction in possible use cases. Nevertheless, it was decided to go forward with the system at hand.

3.3. Memory Partitioning and MPU Registers

With the help of the MPU, the accessible memory range can be partitioned. Those partitions are called regions in the context of this thesis. To control the size and access permissions of those regions, there are two registers required: configuration and address registers.

A single MPU configuration entry has a width of 8-bit, which is densely packed together with other configuration entries into single word-sized (32-bit) configuration registers (cfg_0–cfg_N). This minimises the required number of registers

and the required context-switch time. Each entry together with its corresponding address register forms a configuration tuple, which describes the size and access permission of a region. As described in Section 3.2, the number of registers is not fixed.[1] Instead, the number of possible and available configuration CSRs depends on the required granularity of the address space partitioning and use case for which the MPU shall be used for. Configuration and address registers are modelled in the MoT and the number of registers can be changed freely to accommodate the needs of the target application (see Chapter 4 for details regarding this process). A larger number of registers allows for smaller granularity, flexibility, and more flexible configuration. The layout (i.e., the placement within the CSR address space) of the configuration registers can be seen in Figure 3.1.

The reason to have a flexible number of regions is to accommodate for as many use cases as possible. Some applications might just want to prohibit a single address region from accessing input/output (IO) at all, while others need an elaborate access control scheme.

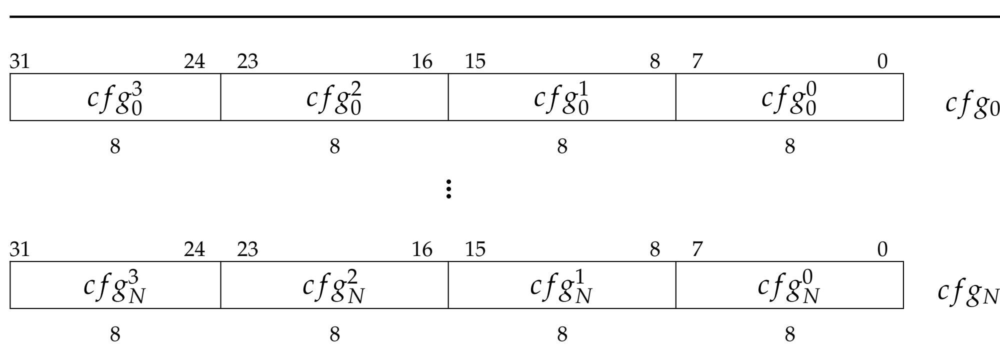

Figure 3.1.: MPU configuration CSR layout (adapted from [WA19b])

The address CSRs hold the base addresses of regions (i.e., the addresses where regions start). The address encoding used for those registers can be seen in Figure 3.2. This format is used for all address registers ($addr_0$–$addr_N$).

[1]The number of registers may only change during the design phase. The number of register cannot be modified after synthesis.

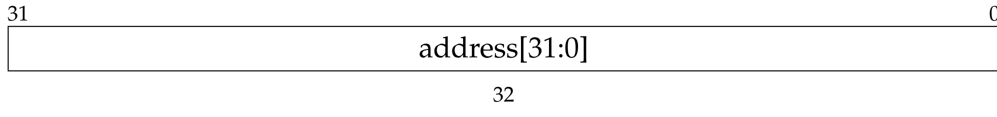

Figure 3.2.: MPU address register format (adapted from [WA19b])

The layout of each configuration entry (cfg) from Figure 3.1 is shown in Figure 3.3. The 'R' and 'W' bits indicate, when set, whether read (i.e., load instructions) or writes (i.e., store instructions) are permitted respectively. Logically this also implies that a cleared bit denies the corresponding access type. A set 'X' bit indicates that an instruction fetch is allowed from the corresponding memory region.
The other fields, 'A', 'B', and 'L' are described in the following sections.

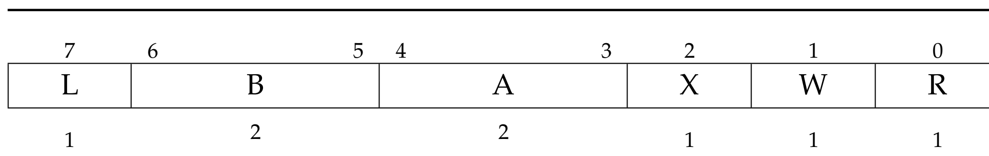

Figure 3.3.: MPU configuration entry (cfg) layout (adapted from [WA19b])

Address and Byte Ranges

As described in the paragraphs before, a region is determined by a configuration entry and its associated address register. The 'A' field in the MPU configuration entries encodes the address-matching mode of a region. This means that this field determines how the address register is interpreted (i.e., the start- and end address of a region and whether it is active). As shown in Table 3.1, there are five matching modes available in total.
The simplest case is $A = 0$. For this case, the entry is disabled, and no matching occurs (i.e., all accesses to this region are forbidden). The four other access modes are:

1. naturally aligned power-of-2 (NAPOT) region;
2. naturally aligned word (4 byte) region (NA4);

3. top boundary of an arbitrary range (TOR) (available only for execution control); and
4. Byte Mode, which replaces the TOR mode for read and write accesses.

A	Name	Description
0	OFF	Disabled region
1	BM/TOR	Byte Mode (for R/W)/Top of Range (for X)
2	NA4	Naturally aligned word (4 byte) region
3	NAPOT	Naturally aligned 2^n byte region, $n >= 3$

Table 3.1.: 'A' field encoding of configuration registers of MPU (adapted from [WA19b])

A more detailed description of each mode is given in the following paragraphs.

These modes support a granularity down to four bytes for instruction memory regions, and down to 1 byte for data memory regions. It should be noted that the assumed four-byte granularity of instruction memory regions is inconsistent with the compressed instruction set specification, which supports 16 bit granularity [WA19a]. However, as mentioned in Section 3.2, the instruction format this work is based on is the RV32I format. In this format, each instruction is exactly 32 bit wide, therefore fulfilling the assumption.

The size of a naturally aligned power-of-2 (NAPOT) region is encoded in the low-order bits (i.e., the least significant bits (LSBs)) of the associated address register. By fixing the LSBs, all possible address variations fall into the specified range. Which means that the address stored in the address register is the encoded start address (i.e., the lowest possible address) of a region. The encoding can be seen in Table 3.2.
It should be pointed out, that this is only possible because the address encoded in the address registers corresponds to bits 33 to 2 of an address. Meaning that the effective address (i.e., the resulting address) will be obtained by logically left-shifting the encoded address by two. The value to store in the address register for a specific region can be calculated: given the start address of the region a, and the supposed size of the region s, the value v to store is

$$v = (a + \frac{s}{2}) \gg 2$$

where $\gg$ denotes a logical right shift.

For regions that use the top of range (TOR) mode, the address range is determined with the associated address register which contains the top of an address range (i.e., the highest address), and the preceding address register which contains the bottom of an address range (i.e., the lowest address). If the 'A' field of a configuration entry i is set to TOR, an address x will fall within the determined range when $addr_{i-1} \leq x < addr_i$. In case there is no preceding register available, 0 will be used for the lower bound, thus, an address x will fall into the region when $x < addr_0$.

addr	cfg.A	Match type and size
xxxx...xxxx	NA4	4-byte NAPOT range
xxxx...xxx0	NAPOT	8-byte NAPOT range
xxxx...xx01	NAPOT	16-byte NAPOT range
xxxx...x011	NAPOT	32-byte NAPOT range
xx01...1111	NAPOT	2^{32}-byte NAPOT range

Table 3.2.: NAPOT range encoding in address and configuration registers of MPU (adapted from [WA19b])

The easiest mode to understand is the naturally aligned word (4 byte) (NA4) mode. When the 'A' field of a configuration entry i is set up to use this mode an address x falls within the range, if and only if, the address matches the value of the associated address register. Formally fulfilling the following requirement $x \stackrel{!}{=} addr_i$.

For regions on which load and store instructions could be executed (i.e., data memory), TOR mode is replaced by Byte Mode (BM). In this mode, the address is matched as an NA4 region; however, the granularity is lowered to byte accuracy. This lower granularity is achieved with the 'B' field in the configuration entries. The encoding of the 'B' field can be seen in Table 3.3.

When the 'A' field of a configuration entry i is set to Byte Mode then the entry matches any address x with the value b from the 'B' field such that $x = addr_i + b$. This approach requires unaligned data memory access support, which was available during the creation of this thesis. Furthermore, it should be pointed out that separate configuration tuples are required for each region to be controlled in BM.

B	Field	Description
0	[7:0]	Access to bit 7 down to 0
1	[15:8]	Access to bit 15 down to 8
2	[23:16]	Access to bit 23 down to 16
3	[31:24]	Access to bit 31 down to 24

Table 3.3.: 'B' field encoding of configuration registers of MPU

Register locking

The 'L' bit indicates whether the configuration entry is locked, and thus writes to the configuration (e.g., cfg_0) and to the associated address register (e.g., $addr_0$) are ignored. When entries are locked, they can only be unlocked through a system reset or reboot.

Priority and Matching Logic

All configuration tuples are not prioritised meaning that all tuples contribute to whether an access succeeds or fails. For an access to succeed, the access address must entirely fall within the range of at least one configured region. Otherwise, the access will fail, regardless of the 'L', 'R', etc. bits. This behaviour can be observed, for example, in case of operations that exceed the protected region, e.g., a half-word (16 bit) read from a Byte Mode protected region which only allows single byte accesses.

If a region is successfully matched, then the 'L', 'R', 'W', and 'X' bits determine whether the access is granted (success) or denied (fail). In case no entry is matched, but at least one region is implemented, the access will always result in a fail. However, in case there are two conflicting matching regions (e.g., one allowing stores and the other one does not) the access will succeed.
Failed accesses generate the respective exception (load, store, or instruction access fault) which must be handled in software through, e.g., an exception handler.

MPU Operation and Status

The MPU operation can be controlled by setting or clearing the ‘E’ bit of the *mpuctrl* register (the layout can be seen in Figure 3.4). No fault will be generated as long as this bit is not set. It should be noted that this bit should be cleared before attempting to change the currently loaded configuration.

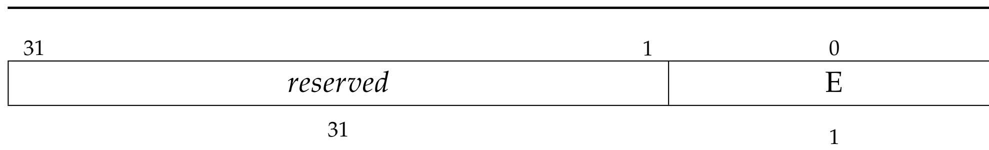

Figure 3.4.: MPU control register format

Figure 3.5 shows the layout of the *mpustatus* register. In this register, the number of currently unconfigured and overall available regions (i.e., the number of configuration tuples) is encoded. This information could be used to ease configuration changes during a context switch.

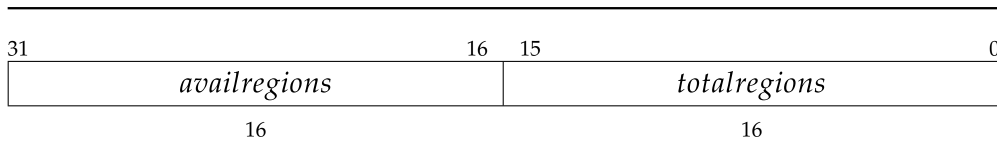

Figure 3.5.: MPU status CSR layout

4. Extension to the CSR model

In the previous chapters, hard- and software assumptions of a module protection unit and the concept of Infineon's HDL generator 'metaRTL' was presented. This chapter focuses on the extension of the CSR model and implementation. Previous work already created a metamodel of CSRs, which made these registers visible in the MoT of the CPU core itself. Prior to the work of Zappia [Zap18] the CSRs were not modelled and directly specified in the implementation, i.e., the MoD layer of the core. While this contribution was sufficient for exception handling, it does not provide the flexibility required by the concept described in Chapter 3. To accommodate the dynamic nature during the creation of the protection unit, several changes and additions where necessary. The focus of this chapter is the implementation of these changes.

To make use of the metaRTL HDL generation flow, the specification (i.e., the metamodel) has to be changed. As described in Section 2.1, the ToD consists of multiple Python scripts which transform the MoT into the MoD. Starting from the MoD, the remaining generation flow is executed automatically, and HDL code is subsequently output, for example, VHDL or Verilog code. This code can then, in turn, be synthesised, and either run on a simulation testbench or a field programmable gate array (FPGA).
For the sake of clarity, the actual Python code in this chapter will largely be omitted, focusing on the structure, while still giving an overview of the CSR implementation and highlighting the changes made.
The remainder of this chapter is split into two parts: 1. in Section 4.1 an overview to the CSR metamodel, the ToD, and thus the implemented classes, is given. Their corresponding hardware architecture is given as well, and; 2. the MPU specific changes are discussed in Section 4.2.

4.1. CSR Template and Model of Design

The CSRs are defined in the MoT of the CPU. They were modelled using the 'StateObject' reference to the 'ObjectProperties' class of the CPU core metamodel. See Figure 5.1 [p. 34] in the following chapter for a depiction.

The actual implementation of the CSRs is specified as discussed above in the Template of Design (ToD). The in the generation involved classes of the MoD in the form of a unified modelling language (UML) diagram is illustrated in Figure 4.1. This section gives an overview of the ToD and a description of each class involved.

CSRBase

This class serves, as the name suggests, the purpose of having a default template for CSR related classes. By inheriting the class attributes, code duplication is reduced.
All CSR related information from the MoT and a reference to the specific MoT instance are stored in variables. Furthermore, an abstract function *getCSR* provides a standardised interface of accessing the underlying CSR structures.

This base class behaviour could not be achieved with the 'CSR' class described later in this section, because of the extended functionality (e.g., the generation of hardware structures) of the aforementioned class.

CSR_RegisterFile

The 'CSR_RegisterFile' class is a structural class (i.e., a class that generates hardware). It reads the MoT and generates CSR instances accordingly. Each CSR object is saved into a dictionary and accessible through the methods provided:

- *autoCSRswiring()* hardwires all unconnected hardware ports of all CSRs to '0'.

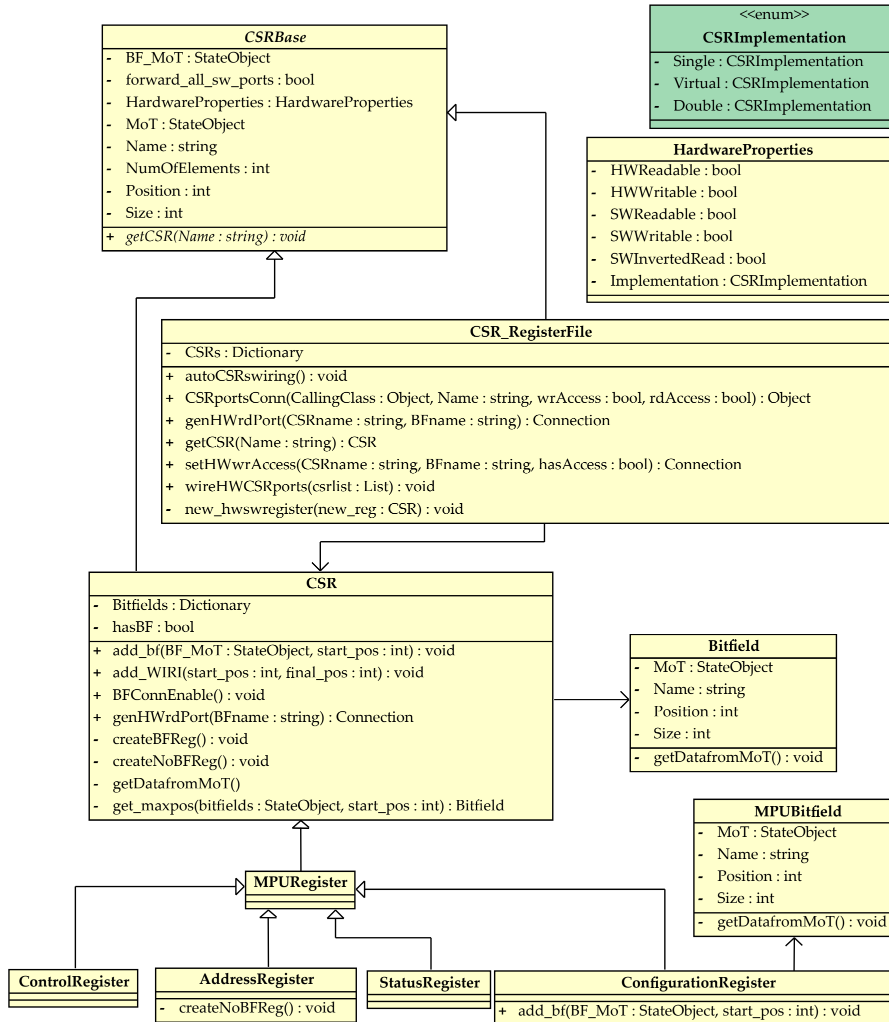

Figure 4.1.: Extended UML diagram of the CSR classes (adapted from [Zap18])

- *CSRportsConn()* generates external ports connected to the hardware ports of the CSR, as they are not generated by default (only the software access ports are). This method generates the hardware read, write, and enable ports for both specified CSR or Bitfield. The generated ports are

subsequently also connected with the ports of the CSR or Bitfield.
By specifying the 'CallingClass' argument, this method's behaviour is the same as just described, but the ports are generated in the 'CallingClass'.

- *genHWrdPort()* helper function of *CSRportsConn()* which creates the hardware read ports.
- *getCSR()* overrides the method from the base class and returns the specific CSR associated to its name.
- *setHWwrAccess()* helper function of *wireHWCSRports()* which creates the hardware write and enable ports or connects the underlying ports of the CSR to '0' in case 'hasAccess' is not specified.
- *wireHWCSRports()* this function creates and connects the required hardware ports to connect them to an external unit (e.g., the exception unit).
- *new_hwswregister*: internally connects the instantiated registers depending on the in the MoT specified access permissions [Zap18].

Figure 4.2 shows the main components of the CSR register file structure (i.e., the equivalent digital logic circuit).[1] The focus of this particular figure lies on the visualisation of the components and ports required to accommodate software (SW) accesses; therefore, the hardware (HW) access ports were omitted.

The main component of this structure is the decoder which takes the write mode (*wr_mode*) and write address (*wr_addr*) to set the enable signal of the specified CSR to HIGH. When the enable signal is set, the data from the write data (*wr_data*) port is written to the CSR. Similarly, data can be read from a particular CSR by setting the read address (*rd_addr*) to a valid value. Both read and write addresses are checked for validity in the lookup table (LuT) of the decoder during the 'decode' stage of the pipeline. The CSR's content can then be read from the read data (*rd_data*) port [Zap18].

The connections between instantiated registers, decoder, and read multiplexer are created if the access permissions were granted in the MoT. Currently, violations of these permissions will not cause any fault that is forwarded to the *error* port (omitted in Figure 4.2 as it is unused), because the CSRs implemented as 'Single' are registers specified in the privileged RISC-V ISA specification by Waterman and Asanović [WA19b], and as such defined as 'write any, read legal (WARL)'. Meaning that all write operations are allowed, although they may

[1] All registers are implemented as clocked and rising edge sensitive; the clock signal input, however, was omitted in all depictions following to improve readability.

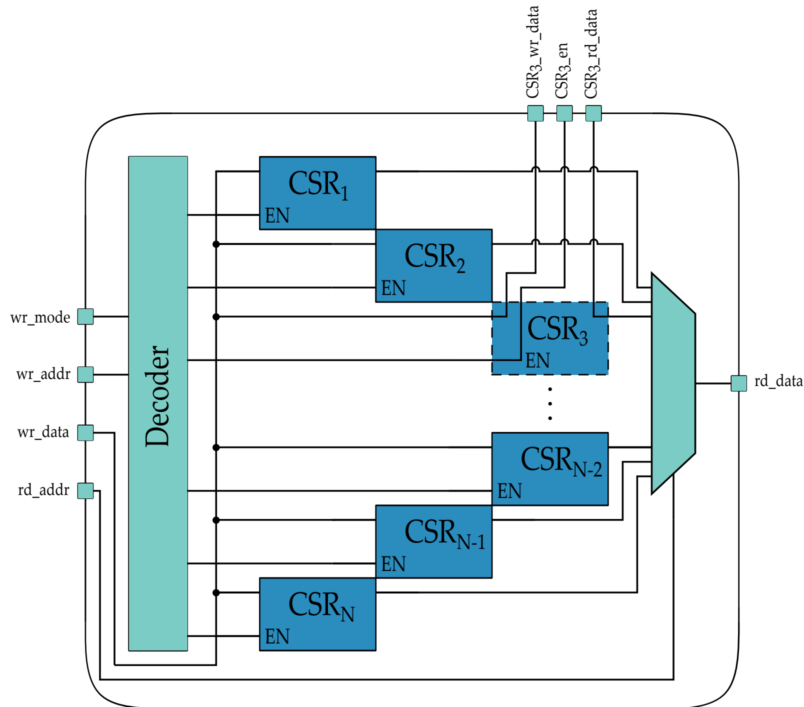

Figure 4.2.: CSR RegisterFile Structure

not necessarily modify any content, and any read operation will return a legal value. Similarly, illegal instructions that occur on 'Virtual' CSRs of the MPU follow the same behaviour.

CSR

In this class, all hardware components are generated during instantiation, and it holds all CSR specific information defined in the MoT.
The hardware components are generated with two methods:

- *createNoBFReg()* is called when there are no bitfields defined in the MoT generating the structure visualised in Figure 4.3.
- *createBFReg()* is called when there are bitfields defined in the MoT. It calls the *addBF()* and *addWIRI()* ('write ignored, read ignored (WIRI)') methods

to instantiate and generate a new Bitfield class or a permanently disabled register respectively. Thus, the overall register structure will consist of multiple smaller internal registers. This can be seen in Figure 4.4 [Zap18].

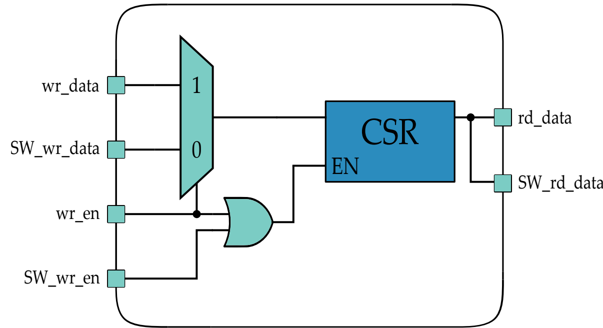

Figure 4.3.: CSR Structure with no Bitfields (adapted from [Zap18])

If there are no bitfields defined in the MoT the *createNoBFReg()* method is called, which generates the structure shown in Figure 4.3.
The structure is quite simple as it consists only of a few components. The input multiplexer selects between the write signal (*wr_data*) of the CPU internal logic and the write signal of the software (*SW_wr_data*) (i.e., access through CSR instructions defined in the ISA). The register is written to if either the hardware enable signal (*wr_en*) or the software enable signal (*SW_wr_en*) is HIGH. However, as the selector signal of the multiplexer is connected to *wr_en* the HW access is prioritised over the SW access in case both enable signals are active [Zap18].

If there are bitfields defined in the MoT the *createBFRef()* method is called. Internally, this method calls the *addBF()* and *addWIRI()* methods respectively, dependent on the order the bitfields were specified in the MoT. The overall register structure will consist of multiple smaller internal registers which can be accessed individually.
An example of a resulting structure is shown in Figure 4.4. For each bitfield, one write and read data port (**_wr_data* and **_rd_data*) as well as an enable port (**_wr_en*) is created for HW accesses. The hardware data ports are concatenated and passed to the input multiplexer, similarly to the CSR without bitfields. The output from the input multiplexer is split so that only the corresponding input data is passed the underlying bitfield register. To facilitate a SW read of

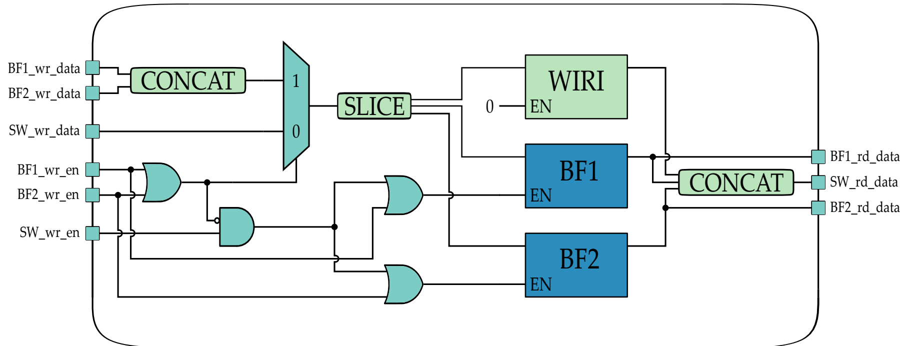

Figure 4.4.: CSR Structure with Bitfields (adapted from [Zap18])

all bitfields at once, the output of all bitfields is concatenated, and its output connected to the SW read port (*SW_rd_data*).
The hardware write priority is achieved with an 'OR' reduction of all HW enable signals that are connected to a negated port of an 'AND' gate. The other port of said gate is connected to the software enable port. This connection makes sure that software access will only happen if there is currently no ongoing hardware access. The signal selection also uses the output of the 'OR' gate. The underlying bitfields are either enabled all simultaneously through a SW access, or individually with a HW access.
In case some bitfields are left undefined in the MoT a WIRI register is generated automatically. Such a register has its enable port hardwired to '0', therefore disabling it permanently [Zap18].

HardwareProperties

This class holds the hardware properties for each CSR.
As seen in the UML diagram in Figure 4.1 these are: access permissions (HW/SW read and write permissions), and implementation mode ('Single', 'Virtual', and 'Double').

'Single' is the default implementation mode, in which case the CSR is instantiated in place as part of the register file structure. This implementation mode is visualised with solid line borders of the CSRs in Figure 4.2. When this mode is used, accesses to the CSR can be made as described previously. All registers of

the exception unit are implemented using this mode (as seen in [Zap18]).
'Virtual' is functionally identical to 'Single' in terms of accesses; however, the actual instantiation of the CSR is done externally (i.e., the register is part of some other structure). When this mode is used, the read, write, and enable ports are created on the top level of the register file structure. The ports will be automatically connected externally during the pipeline creation. For this to happen, the ports at the register file top level must have the same name as the corresponding ports on the external structure. Internally, these ports are connected to decoder and read de-multiplexer the same way 'Single' registers are.
This extension to implementation modes allows having various CSR instances (of multiple CSR classes), which, for example, could be structured differently internally than the default implementation. This way, a backwards-compatible interface is providing allowing accesses to happen as if they were 'Single' CSRs.
This implementation mode was used for all MPU registers and can be seen in Figure 4.2 of CSR_3 (the register with a slotted line).
In 'Double' mode, a CSR bus interface is added to the register file top level. All accesses to and from an address range specified in the MoT are routed to this interface. This approach has similar benefits as the 'Virtual' mode. One advantage of this mode is that the number of ports is reduced because access ports are only added for address ranges instead of individual registers. However, the structure with the counterpart bus interface and CSR instances needs to implement a decoder, because the outgoing bus interface at the register file structure is connected directly to all in-ports (*wr_mode*, *wr_addr*, etc.). This implementation mode is not shown in Figure 4.2, as it was not used for this thesis. It was, nevertheless, created for possible future work.

The importance of the access permissions was already explained previously in the CSR register file class description; as such, it will be left out from this section.

4.2. MPU Extension

This section is about the required extensions to the CSR ToD to ease the hardware implementation of the requirements described in Chapter 3.
As in the previous section, this section gives an overview of the ToD and a description of each class involved.

MPURegister

The 'MPURegister' class is used as a parent class for all inheriting classes ('ConfigurationRegister', 'AddressRegister', etc.). It has the same functions and attributes as its parent. The only difference being the initialization routine, as this class does not differentiate between various implementation modes. This can be left out because it assumed that all registers are instantiated as 'Virtual'.

AddressRegister

This class implements, as the name implies, a register that holds a 32-bit address. It uses the same structure and layout as regular CSRs without bitfields but has an extra port for a lock signal. This signal is input from the lock field ('L' field) from the corresponding configuration, as explained in Section 3.3.

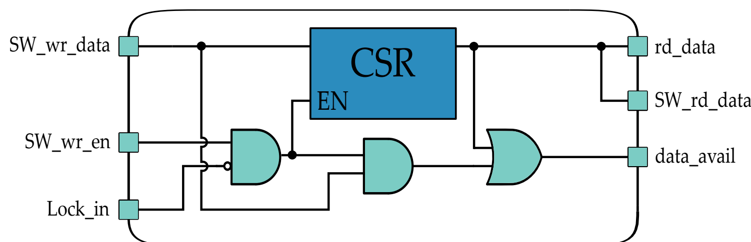

Figure 4.5.: Address Register Structure

As seen in Figure 4.5, this locking mechanism is implemented by connecting the lock signal to a negated in-port of an 'AND' gate. The other input of this gate is connected to the 'OR' reduction of both enable ports, and the gate's output is connected to the enable port of the register. That way, the register will be permanently disabled when the lock signal is set to HIGH. This locked state stays active until the lock signal goes back to LOW.

Furthermore, a port which shows whether data will be written to or is already available was added as well (*data_avail*). This port is used as input for the MPU status register, allowing atomic updates.

ConfigurationRegister

The in Section 3.3 described configuration entries are encoded in the class 'ConfigurationRegister'. This class is a variation of a CSR with bifields, as visualised in Figure 4.6. It should be noted that said figure only shows the structure of a single underlying bitfield, a fully filled register can hold up to four of such configurations in parallel.

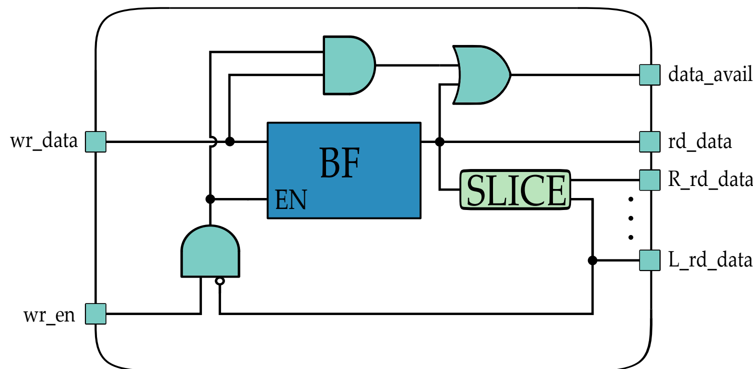

Figure 4.6.: Configuration Register Bitfield Structure

One modification to the configuration entry register ('Bitfield' class) was necessary to implement the locking mechanism. As soon as the 'L' bit is set, the register will lock itself automatically. This is done by permanently disabling the enable port of the register with the help of an 'AND' gate. A negated port is connected to the lock bit, and the other port connected to write enable signal (*wr_en*). The class that implements this change is called 'MPUBitfield'. This class is not a generalisation of the 'Bitfield' class, because it would have caused errors during the instantiation, because of an inheritance problem.

Similarly, to the 'AddressRegister', a *data_avail* port was added as well, for the same reason already described.

StatusRegister

This class is used to store the current and total available number of configurable MPU regions. No specific changes were made in terms of structure, as such a

figure was omitted from this thesis.

ControlRegister

This class is used to control the MPU operation, i.e., whether it is enabled or not. The underlying structure uses a default CSR with bitfields layout.

5. MDA based MPU generation

In the previous chapters, the model-driven architecture (MDA) and the required extension to the CSR model were presented. This chapter focuses on the generation of an MPU within Infineon's MDA framework (see Section 2.2 for details).

5.1. MPU Placement in the RISC-V Model of Things

In this section, an overview of the core metamodel used for this work is given. The original metamodel by Schreiner [Sch16] was extended with a new node ('MPU') to allow for the modelling of an MPU. This extension is the centrepiece of the subsequent sections and will be described thoroughly. All other parts of the model will be described just briefly, as they are out of the scope of this thesis. A very detailed description and discussion of these parts can be found in [Sch16].

Figure 5.1 shows a UML class diagram of the CPU metamodel. This model can be split into five component groups. The added node 'MPU' will be discussed in more detail in the paragraphs following. The other aforementioned component groups are:

- *Architectural State* contains all stateful elements of the CPU that a compiler needs to be aware of (e.g., registers, memories, and flags).
- *Encoding Tree* 'describes how individual instructions are encoded in the instruction words the CPU can execute. This also includes the description of how parameters such as register addresses and immediate operands are encoded in the instruction word.' [Sch16, p. 46]
- *Instruction* models the instructions and the instruction behaviour, i.e., how the architectural state of the CPU is influenced by instructions.

- *Exception Node* models which exceptions can occur and how those influence the architectural state of the CPU [Zap18].

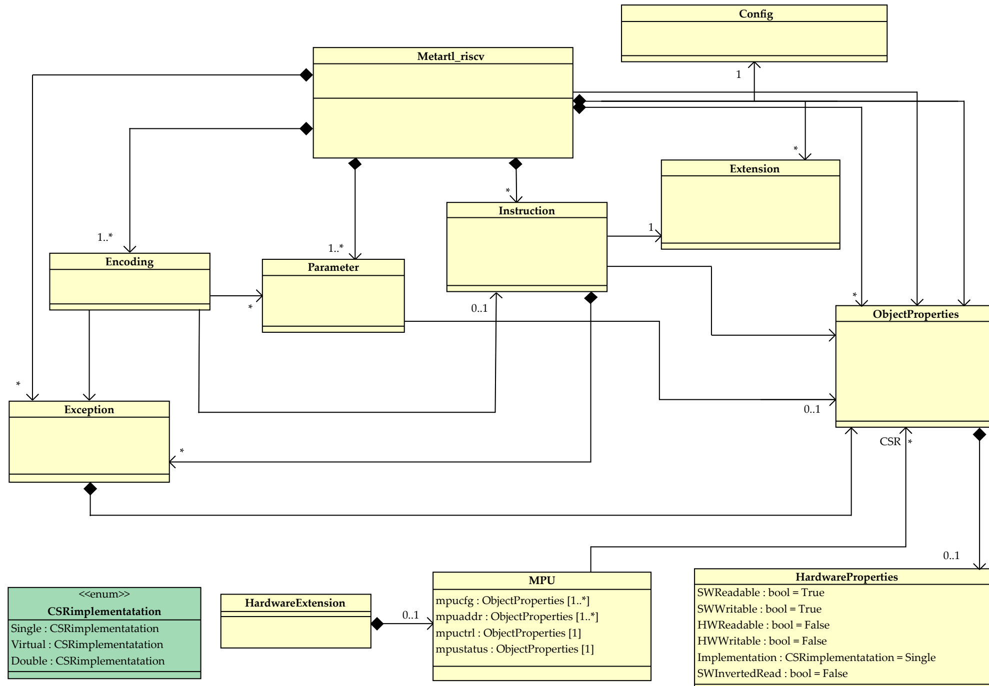

Figure 5.1.: Schematic metamodel of RISC-V core (MetaRISC) expanded with MPU

As depicted in Figure 5.1 the 'MPU' node was modelled as part of a 'HardwareExtension' in such a way that the system may, or may not, have exactly one MPU. The MPU model itself only consists of the registers described in Section 3.3. As these registers are implemented as CSRs, a reference between the MPU and the 'ObjectProperties' class is necessary to keep the resulting metamodel (the MoT) consistent.

5.2. MPU Template and Model of Design

The MPU implementation is specified, similarly to the CSRs described in Section 4.1, in the Template of Design (ToD). A UML diagram of the MPU related classes of the MoD is shown in Figure 5.2. The implementation is specified in the ToD that is part of the MoD layer, as described in Section 2.1. This section gives an overview of how the ToD of the MPU is constructed and a description of each class involved.
For the sake of clarity, in this section the actual Python code (i.e., the code describing the ToD) will be omitted, focusing on the resulting hardware structures.

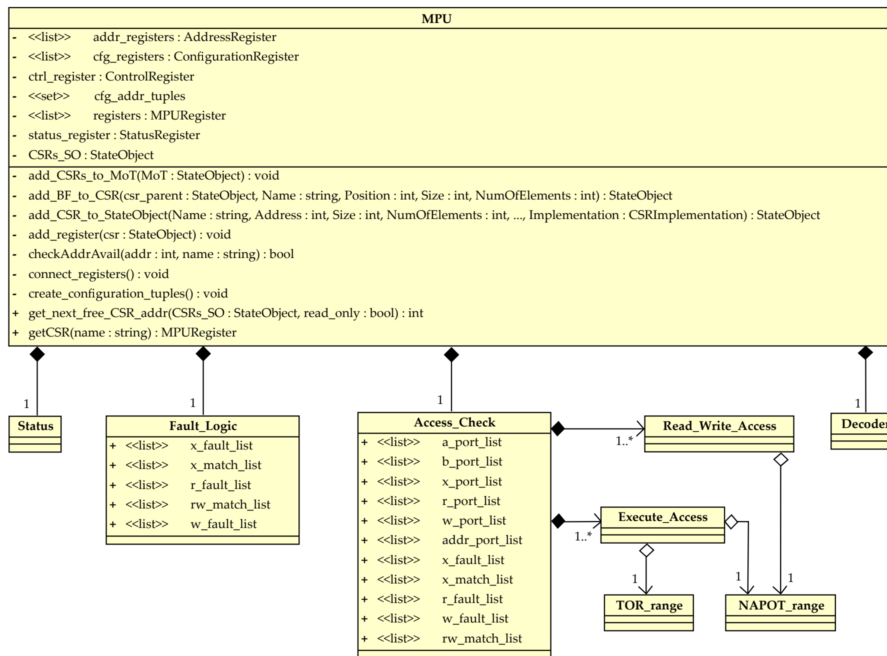

Figure 5.2.: UML diagram of the MPU classes

MPU

The 'MPU' class is a structural class (i.e., a class that generates hardware) which reads the MoT and generates the MPU instance accordingly. The resulting structure is the MPU toplevel, whose ports are connected to the exception unit and placed within the CPU pipeline (see Section 5.3 for details). A depiction of the MPU toplevel structure is shown in Figure 5.3.

The responsibility of this class is, apart from serving as MPU toplevel, to instantiate the required (sub-)structures as well as to modify the core MoT. With this modification, the MPU registers are added as CSRs. Finally, the MPU has to function as a CSR manager. A number of lists, tuples, and dictionaries are used to keep track of the instantiated structures and their ports. These objects are then used by both the MPU class itself and its child components to, for example, create connections between said ports. This is achieved with the following methods:

- *getCSR()* overrides the method from the CSR base class (see Section 4.1 for details), and returns the specific MPU-CSR associated to its name.
- *add_CSRs_to_MoT()* adds, as the name implies, the specified CSRs from the MPU to the MoT instance. This function does this by reading the register information from the MoT. The register data from the MoT includes the register name and size. The CSRs are then added depending on the data read. For example, a configuration register in the MoT with size 3 and the name 'mpucfg0', will be added as CSR with the specified name and its three configuration entries are added as bitfields.
- *add_BF_to_CSR()* is a helper function of *add_CSRs_to_MoT()*. It adds the required bitfields to a CSR. For example, the bitfields of a configuration entry ('A', 'B', 'R', 'W', etc.).
- *get_next_free_CSR_addr()* another helper function of *add_CSRs_to_MoT()*. This function returns the next available address (i.e., an address where no other CSR has been placed) from the custom CSR address range as specified in the privileged RISC-V ISA specification by Waterman and Asanović [WA19b].
- *add_CSR_to_StateObject()* this function is used by *add_CSRs_to_MoT()* to add a specified CSR to the MoT. As described in Section 4.1 the CSRs are modelled as StateObjects, therefore the MPU registers have to be added

to the CSR StateObject.

- *add_register()* instantiates the register structures. The reference to each register instance is saved to the corresponding list. For example, an address register instance is saved to the *addr_registers* list.
- *checkAddrAvail()* this function is used by *add_CSR_to_StateObject()* to verify that there is no address collision (i.e., multiple CSRs at the same address).
- *create_configuration_tuples()* generates a list of tuples in the form $<cfg_0, addr_0>$, ..., $<cfg_N, addr_N>$ to easily find the corresponding address register to a configuration entry (and vice versa). This list is used extensively throughout the generation process of the MPU.
- *connect_registers()* this function has multiple responsibilities. Most importantly it sets up the required connection between the registers, e.g., the 'L'-bit of a configuration entry to the lock port of the corresponding address registers.
  Furthermore, it connects the unused hardware ports of the CSR instances to '0' in order to permanently disabling them. This is necessary because some hardware ports (e.g., *hw_wr*) are a byproduct of the CSR instantiation. These ports are unused but need to be connected because unconnected ports would generate errors during synthesis.
  Lastly, this function also is in charge to make the connections between the other instantiated components, e.g. the connection between the fault logic and access check component.

Figure 5.3 shows an overview of all instantiated components that are generated by the MPU. Each of these components will be described in detail in the sections following. It should be noted that the registers (the block with the dashed line) are not single entities (as shown in Figure 5.3), but rather multiple individual structures. They are depicted that way because it helps to clean up the figure, and they do not add any necessary additional information to the overall picture.

As depicted in Figure 5.3, several external ports are added during the generation of the MPU. The most important ports are the register source data (*rs1_data*), the instruction currently being executed (*instruction*), and the address of the instruction currently being fetched from (*PC*, i.e., the program counter (PC)). The data from those ports is taken in and evaluated by the underlying instantiated components. These components then generate fault

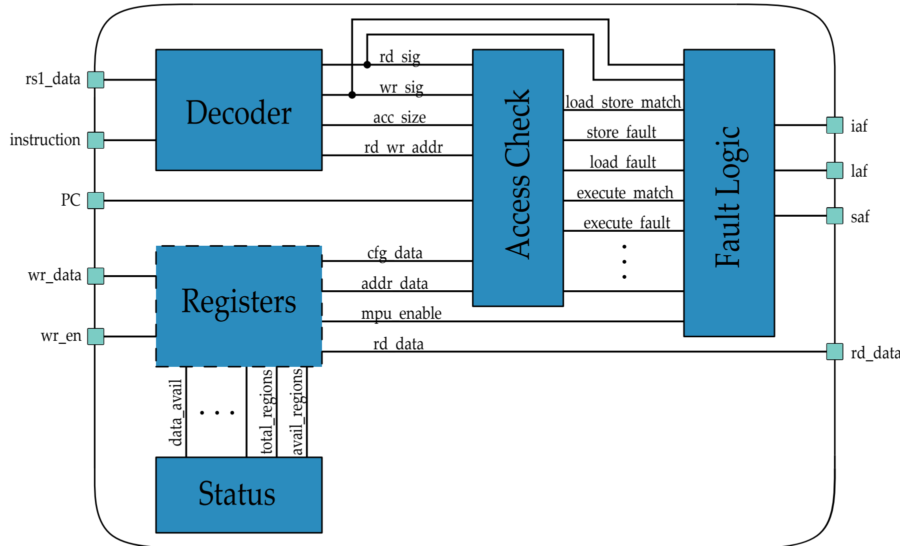

Figure 5.3.: MPU Top Structure

signals in case unallowed accesses occur. The resulting fault signals are connected to the respective fault ports (instruction access fault: *iaf*, load access fault: *laf*, and store access fault: *saf*).
The remaining ports are necessary to allow accesses to and from the registers. Each register has its own read data (*rd_data*), write data (*wr_data*), and enable (*wr_en*) port. Those ports are then connected to the CSR_RegisterFile structure described in Section 4.1.

Status

This class encodes the number of possible protection regions as well as how many of those are currently available (i.e., unset regions). The output ports of the corresponding structure (*regions_avail* and *regions_total*) are connected to the write data ports of the equally named bitfields that are part of the 'mpustatus' register. A depiction of the resulting structure can be seen in Figure 5.4.

The total number possible protection regions (*regions_total*) is set to a literal (i.e., a constant) during the generation of the MPU. The currently unset regions

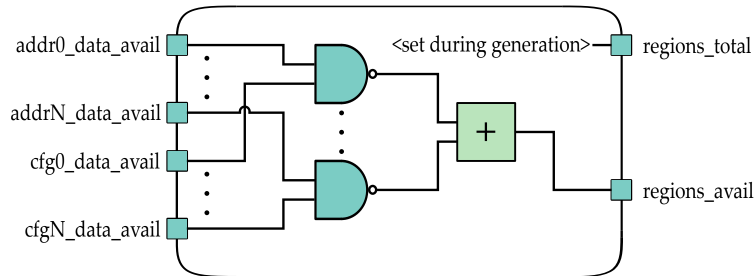

Figure 5.4.: MPU Status Structure

(*regions_avail*) are updated every clock cycle. The available data ports from the address and configuration registers (*addr*_data_avail* and *cfg*_data_avail*) are used as input to 'NAND' gates. Each configuration tuple (address and configuration) has its gate. A region is only used when both configuration and address register are set to a value greater zero. Which means, in turn, that a region is unset if the opposite is true. Therefore, a negated 'AND' or 'NAND' gate has to be used. The output of these gates is then added. The sum is the number of regions that are currently unset.

Decoder

The decoder is used to decode the current instruction and register data to output whether relevant access (read or write) is occurring and its effective address. A depiction of the generated structure to do this can be seen in Figure 5.5.

The main component of the decoder is the slicer which separates the instruction applied to its port (*inst*) into individual signals.
The load and store ports (*rd* and *wr*) are set to HIGH in case the opcode is set to the corresponding value. This is done with the help of a LuT. The load and store signals are used to determine whether the access size (*acc_size*) is output or set to zero. Similarly to the read and write signals a LuT is used to get the current access size encoded in the instruction 'funct3' (as defined in the unprivileged RISC-V ISA specification by Waterman and Asanović [WA19a]). The access size is only yielded when either a load or store occurs. As such, those two signals are used as input to an 'OR' gate. The output of this gate is used as a select signal of the multiplexer which switches between the value of

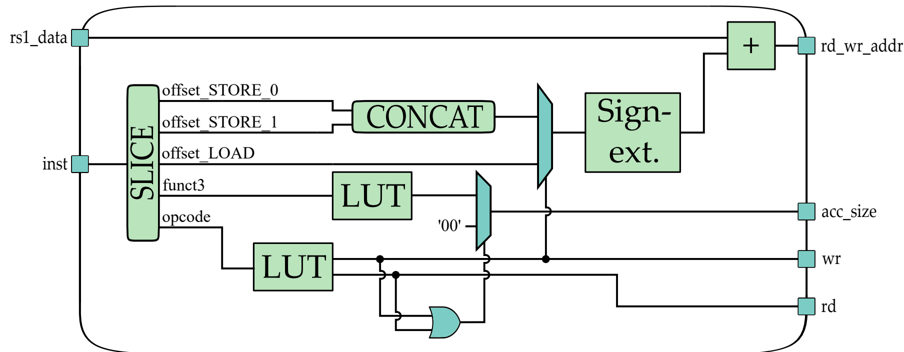

Figure 5.5.: MPU Decoder Structure

the LuT and a default value (i.e., '0').
The write signal is also used as a select signal of the offset multiplexer to select the correct offset. The address offset in case of a store instruction needs to be concatenated first since it is not encoded continuously within the instruction. After the selection process, a sign extension in accordance with the unprivileged RISC-V ISA specification is required in case bit 11 of the offset is set to '1'. In any case, the 12-bit offset is extended to a 32-bit value which is added to the value applied to the register data port (*rs1_data*). The output of this operation is the effective access address. This output is directly connected to the output port (*rd_wr_addr*).

The previously described structure is only required for the decoding of load or store (i.e., read or write) instructions, and not for the address of the current instruction. The effective access address of execute accesses is kept track with the PC and as such does not need further processing.

TOR_range

The 'TOR_range' class generates the structure for the top boundary of an arbitrary range (TOR) address matching mode described in Section 3.3. As seen in the UML class diagram in Figure 5.2, this class is instantiated only as part of the 'Execute_Access' class, since this mode is only supported for execution control. A depiction of the TOR structure can be seen in Figure 5.6.

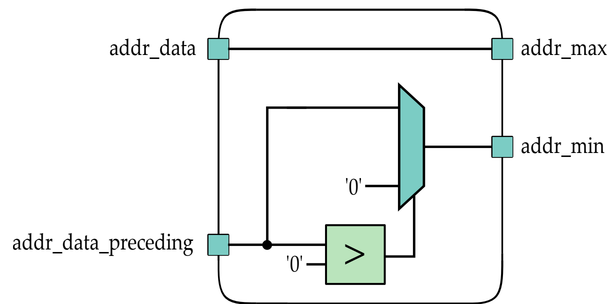

Figure 5.6.: MPU TOR Structure

The structure to accommodate the behaviour described in Section 3.3 is straight forward. The maximum address port of a range (*addr_max*) corresponds to the address data input (*addr_data*). The minimum address should correspond to either the address data from the preceding address register *addr_data_preceding* or '0' in case no preceding address register exists. This behaviour is modelled with a multiplexer and 'GREATER-THAN' comparison. The output of the comparator – which checks whether the preceding address is greater than zero – is used as a select signal of the multiplexer. This multiplexer switches between the preceding address and '0'. The output of this multiplexer is the minimum address port of a range (*addr_min*).

Using this structure, an address x is valid (i.e., it falls within the specified range) if it fulfils $addr_min \geq x < addr_max$.

NAPOT_range

The 'NAPOT_range' class generates the structure for the naturally aligned power-of-2 (NAPOT) address regions described in Section 3.3. As seen in the UML class diagram in Figure 5.2, this class is instantiated by both 'Execute_Access' and 'Read_Write_Access' class. A depiction of the NAPOT structure can be seen in Figure 5.7.

As described in Section 3.3, the size of a NAPOT range is encoded in the LSBs of the address register data. The structure shown in Figure 5.7 is the hardware equivalent of an 'IF-THEN-ELSE' structure found in software. The LSBs of

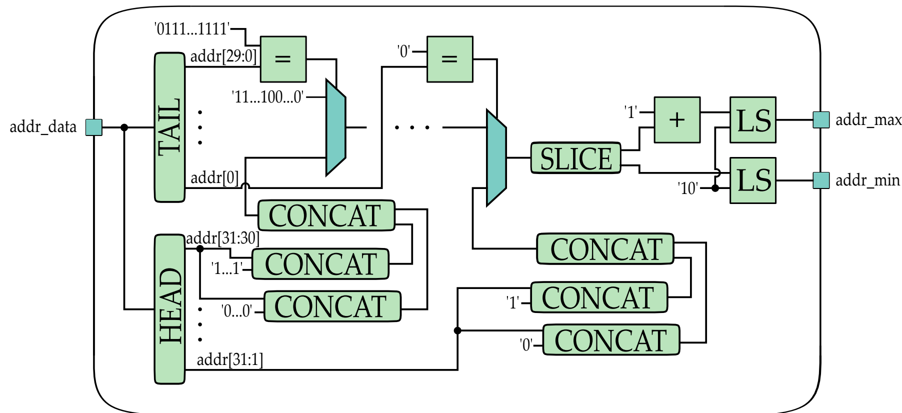

Figure 5.7.: MPU NAPOT Structure

address data are checked with a greedy algorithm. Meaning, the first match found is also the final output.
To do so, the data applied to address data port (*addr_data*) is first split into two sections: head and tail section. The tail section corresponds to the LSB to be checked, while the head section encodes the base address of the NAPOT address range.[1] The head section is concatenated with both '0"s and '1', to get both the minimum and maximum address of a range respectively. The output of those two concatenations is then also concatenated for further processing.
A chain of multiplexers is used to determine the correct LSB match. Each multiplexer has two inputs: the first input is connected to the output of the previous multiplexer, and the second input is the output of the aforementioned concatenation. This chain is continued until all 30 possibilities are exhausted.[2] The last multiplexer (i.e., the leftmost multiplexer in Figure 5.7) has its default input port connected to the theoretical maximum range (2^{32}-byte range) and the other the corresponding concatenation. To select the correct multiplexer input the tail sections are compared to predefined LSB masks. Each of the results of these comparisons is used by a matching multiplexer as a select signal. Once

[1]It should be noted that all possible head and tail sections are created in parallel to allow for parallel checking in hardware. This also means that the size of these sections is not constant (a greater tail section means a smaller head section, and vice versa).

[2]The in- and outputs of the multiplexers form a chain; however, the selection ports are activated simultaneously (i.e., in parallel). Therefore, a parallel lookup is performed.

the matching bitmask was determined further processing is applied to the output of the final multiplexer (i.e., the rightmost multiplexer in Figure 5.7) to get the resulting address range from the address data input.
First, the multiplexer output is split in half; the lower 32 bit corresponding to the lower bound of the range, and the other 32 bits corresponding upper bound. The upper bound is further incremented by 1 for easier address checking. Finally, both bounds need to be logically left-shifted (i.e., the most significant bit (MSB) is shifted off, and the LSB set to '0') by 2 to correct the boundaries. The left-shift is necessary, as described in Section 3.3, because the address encoded in the address registers corresponds to bits 33–2 of a physical address. The outputs of those shift operations are connected to the maximum address port (*addr_max*) and the minimum address port (*addr_min*) respectively.

Using this structure, an address x is valid (i.e., it falls within the specified range) if it fulfils $addr_min \geq x \leq addr_max$.

Access_Check

The 'Access_Check' class is used as a top module to instantiate the check structures for both read/write ('Read_Write_Access') and execute ('Execute_Access') accesses. Both 'Read_Write_Access' and 'Execute_Access' are described in more detail in the sections following. This class instantiates and connects each of the structures as mentioned earlier for every possible MPU protection region. The outputs of those structures will be forwarded to the 'Fault_Logic' structure, which is also described in a later section. A depiction of the resulting structure can be seen in Figure 5.8.

As seen in Figure 5.8, ports for each region (configuration entries (*cfg*_data*) and address (*addr*_data*) data) are added to the toplevel. These ports are used by both 'Execute_Access' and 'Read_Write_Write' structure, to generate the respective fault (*EXECUTE_fault*, *STORE_fault*, and *LOAD_fault*) and match (*EXECUTE_match* and *LOAD_STORE_match*). The former speaks for itself and does not need further explanation. The latter signals whether the current access address (*PC* or *rd_wr_addr*) falls within the validity range of the underlying structure.
The 'Execute_Access' structures do not require any additional signals to function, the 'Read_Write_Write' structures, however, require also the memory

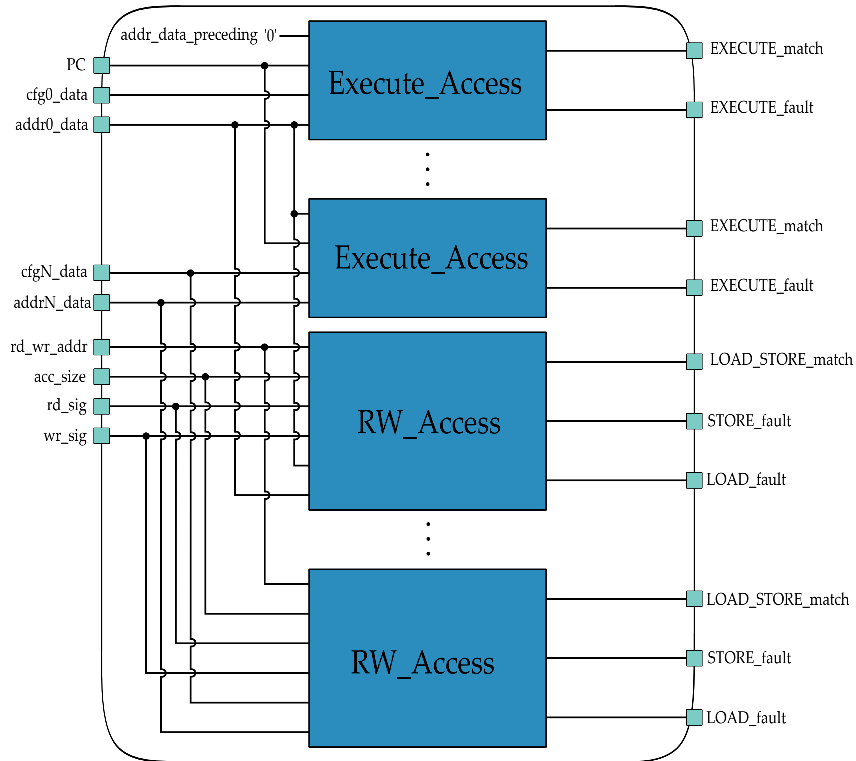

Figure 5.8.: MPU Access Check Structure

access size (*acc_size*) and the information on if a load or store is currently being executed (*rd_sig* and *wr_sig*). These signals are not necessary for the 'Execute_Access' structures because of the assumptions made in Section 3.2. As described in Section 3.2 all instructions are assumed to be word-aligned and 4-byte wide, and as such no access size information is required. Furthermore, an execute access happens on every PC change. Therefore, an execution signal can be left out.

Execute_Access

The 'Execute_Access' class generates the structure to check whether the execution of an address causes an instruction access fault. As seen in the UML

diagram in Figure 5.2 this class is instantiated as part of the 'Access_Check' class previously described.

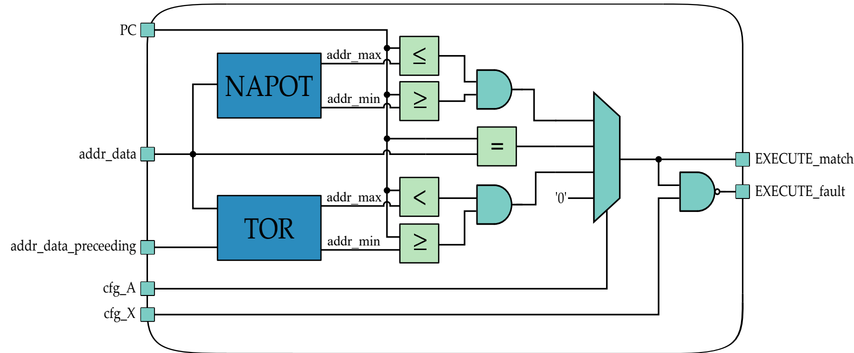

Figure 5.9.: MPU Execute Access Structure

The centrepiece of this checking structure, as seen in the depiction of the resulting structure in Figure 5.9, is a multiplexer. This multiplexer switches between its inputs depending on the matching mode field ('A' field, as described in Section 3.3) from a configuration entry (*cfg_A*). To these inputs, the corresponding matching structure (NAPOT, NA4, and TOR) is connected. The default input port is connected to '0'. The current access address (*PC*) is checked to fall within the range determined by the output the matching structures (i.e., minimum and maximum address). If it falls within the range, the output of the 'AND' gate will be HIGH. The output multiplexer is directly connected to the match port (*EXECUTE_match*), but also to an 'NAND' gate which verifies whether this operation is permitted. The execution fault port (*EXECUTION_fault*) will be set to HIGH in all but one cases. Namely, in case the execution bit ('X' field, as described in Section 3.3) from the configuration entry (*cfg_X*) is set and the execution address (*PC*) was matched successfully. In all other cases (no match, no execution permission, or both) for a given address the fault signal will be set.

Read_Write_Access

The 'Read_Write_Access' class generates the structure to check whether a load or store to or from an address causes a load or store access fault. As

seen in the UML diagram in Figure 5.2 this class is instantiated as part of the 'Access_Check' class previously described.

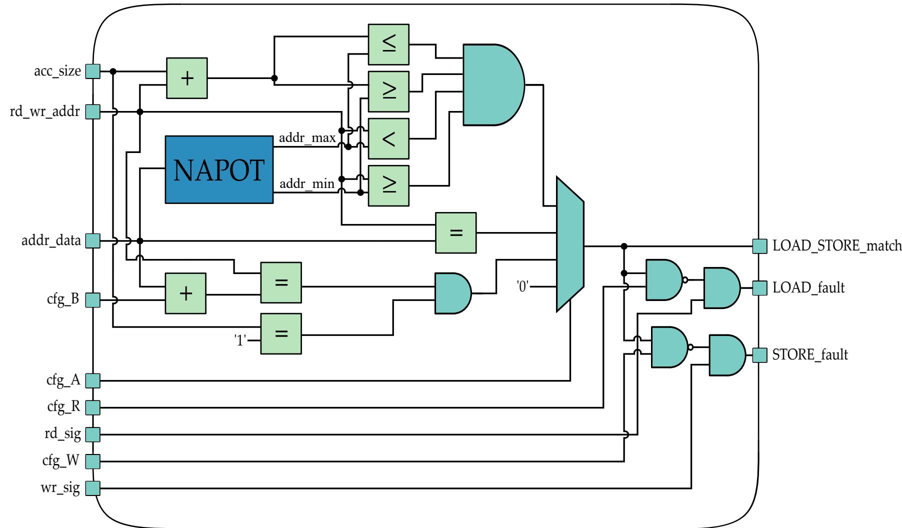

Figure 5.10.: MPU Read/Write Access Structure

The structure utilises the same idea as the 'Execute_Access' structure. The only notable difference is the inclusion of the Byte Matching (BM) mode described in Section 3.3. In this mode the access address (*rd_wr_data*) must match the address encoded in the address register (*addr_data*) plus the byte offset field ('B' field, as described in Section 3.3) from a configuration entry (*cfg_B*)).
Another difference lies in how the NAPOT range is checked. Load and store instruction can access unaligned memory. Therefore, the case of an access 'crossing' a word boundary needs to be caught. For example: given an address region with the range $0x1000$–$0x2000$, a word-sized load occurs on address $0x1FFD$. Such an operation would load 3-byte from the specified region, but also 1-byte from the adjacent region. Given this case, a fault has to be generated since the access does not fit completely inside the specified region. The same is true for the same sized load on, e.g., address $0xFFE$. In which case, the same behaviour can be observed. To do this, the access size (*acc_size*) is added to the address to be accessed (*rd_wr_addr*). The result of this addition is then checked if fully falls with the NAPOT range.
The fault signal generation is also similar to the 'Execute_Access' structure. The

difference being that load and store instructions (indicated by their respective signal *rd_sig* and *wr_sig*) need to happen for the fault signal (*LOAD_fault* or *STORE_fault*) to be set to HIGH. The matching signal (*LOAD_STORE_match*) will be set regardless of the current regions load or store permission ('R' and 'W' fields, as described in Section 3.3) set in configuration entry (*cfg_R* and *cfg_W*).

Fault_Logic

This class generates the structure that is ultimately responsible for checking whether an access fault occurred.

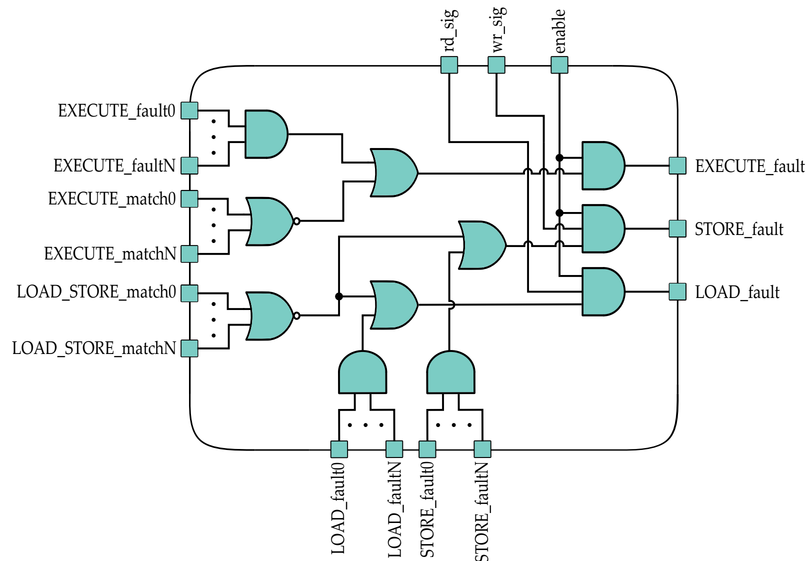

Figure 5.11.: MPU Fault Logic Structure

As seen in Figure 5.11, the toplevel has connections for each match and fault signal (*_match* and *_fault*) from the 'Access_Check' structure. A fault occurs in two cases: 1. no match was found (inputs to 'NAND' gates); or 2. all access structures found a permission violation (inputs to 'AND' gates).
The output of the three 'OR' gates (one for each fault type) is used as input for the final 'AND' gates. The other input is the MPU enable port (*enable*).

Which, when HIGH, means that the respective fault will be routed to the outport (*EXECUTE_fault*, *STORE_fault*, and *LOAD_fault*). In order for load or store faults to happen, the respective instructions need to happen which are indicated by the signals *rd_sig* and *wr_sig*.

5.3. MPU Placement within the CPU Pipeline

The MPU structure described in Section 5.2 has several ports that need to be connected within the 5-stage CPU pipeline. This section discusses the MPU placement and connection to other components within the CPU pipeline.

The pipeline design used for the core this work is based on is a classical 5-stage reduced instruction set computer (RISC) design with the following stages:

1. *Instruction Fetch (IF)* loads the instruction from the instruction memory pointed to by the program counter (PC).
2. *Instruction Decode (ID)* decodes the fetched instruction.
3. *Execute (EX)* executes the decoded instruction.
4. *Memory (MEM)* accesses data memory if required.
5. *Write Back (WB)* in this stage the results are written into the register file.

To catch possible illegal accesses as early as possible, the MPU needs to be placed as early as possible in the pipeline. The first stage that the PC is not modified is the ID stage. It should be noted that it would be possible to place the MPU also in the IF stage; however, this would lead to a pipeline flush in case of an execution fault. This is due to the behaviour of the exception unit when an exception is raised during an instruction fetch. The way the exception unit was designed an instruction access fault (i.e., an execution fault) is only supposed to occur during the ID stage. If such an exception is raised during the IF stage, this leads to a pipeline flush. Details regarding this behaviour can be found in the work of Zappia [Zap18]. Keeping this behaviour in mind, it was decided to place the MPU in the ID stage to reduce the number of pipeline flushes and therefore increase the system performance.

The MPU port connection is straight forward: the PC is connected coming from the exception unit in the IF stage (i.e., the to the ID stage forwarded PC port); the fault ports are connected to the pre-allocated fault ports of the exception

unit; the register read and write ports are connected to corresponding ports of the CSR_RegisterFile placed in the EX stage; and finally, the register data port (*rs1_data*) and the instruction port are connected to their respective counterpart in the EX stage as well.
The last two mentioned connections need to come from the execute stage because the register file content can change between the ID and EX stage. This change of data can lead to wrong offsets and spurious faults when checking for illegal addresses in the ID stage, therefore a delay until the next pipeline stage is required.

6. Evaluation and Results

6.1. Behavioural Tests

The behaviour of the generated view (i.e., HDL source code) of the MPU described in Chapter 5 was tested with XSim included in Xilinx Vivado v2018.2. The behaviour of individual logic circuits (e.g., the resulting structure of the decoder) was checked with manually implemented VHDL testbenches. The resulting waveform of the decoder testbench can be seen in Figure 6.1. In these testbenches, the input ports of the unit under test (UUT) were connected to signals with predefined values at a specific time. To check whether the unit behaves as expected, the output ports of the unit were asserted against the expected values.

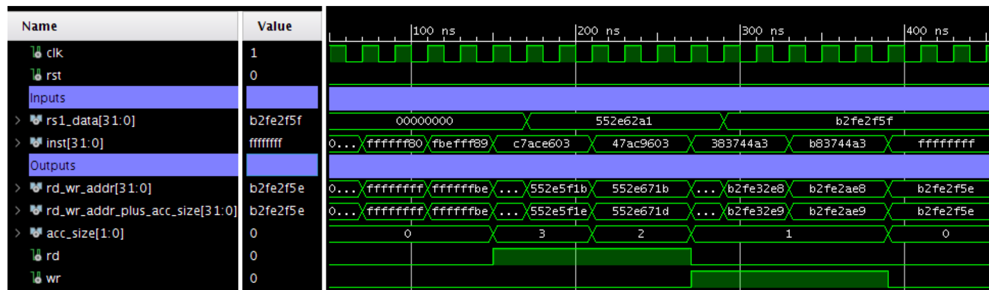

Figure 6.1.: Decoder test

Small test programs were written to test the overall MPU behaviour as part of the RISC-V core. The source code of those programs was compiled to binaries with the GCC 7.1.1 for RISC-V. The binaries were then loaded into the instruction memory of a simulated SoC. In the following sections, the results of the two most significant simulations are presented.

Exception Handling

Figure 6.2 shows the waveform of the signals involved in the interaction between MPU and exception unit. For this test, a small kernel with an exception handler was written. The source code for this test can be found in Appendix A. The task of the kernel and exception handler is to check whether an exception occurred because of illegal access, and, if so, to return from the current function that made an illegal access to the caller. This behaviour simulates the aborting of a task or function (e.g., in an RTOS).

To test the aforementioned behaviour, the MPU is configured to throw exceptions when certain functions try to access specific memory regions. After the configuration of MPU and exception unit, these functions are called one after the other by the *main()* function.
The first function causes an instruction access fault (blue *iaf* signal in Figure 6.2). The fault signal is received by the exception unit (SEU_rq), which then saves the address of the offending instruction – i.e., the PC – to the machine program counter exception registers ($mepc$). Next, the exception unit jumps to the kernel entry by overwriting the PC with the address saved in the trap-handler base address register ($mtvec$). The kernel entry flips an unused bit on the stack (769) to mark the kernel entry and exit. After the kernel entry has saved the current task context – i.e., register values, stack pointer, etc. – the exception handler is called. This handler then sets a region on the stack (770) to predefined values depending on which fault caused the exception. In case of an instruction access fault said region is set to $0xF$. Finally, the task context is restored[1], and the machine program counter exception registers is loaded with the return address to the original calling function (e.g., the *main()* function). Doing so simulates the aborting of a function call.

The other fault types are tested in a similar fashion; therefore, a closer discussion is omitted from this thesis.

[1]It should be noted that instead of restoring the current task context a task switch could occur at this point by restoring a different task context.

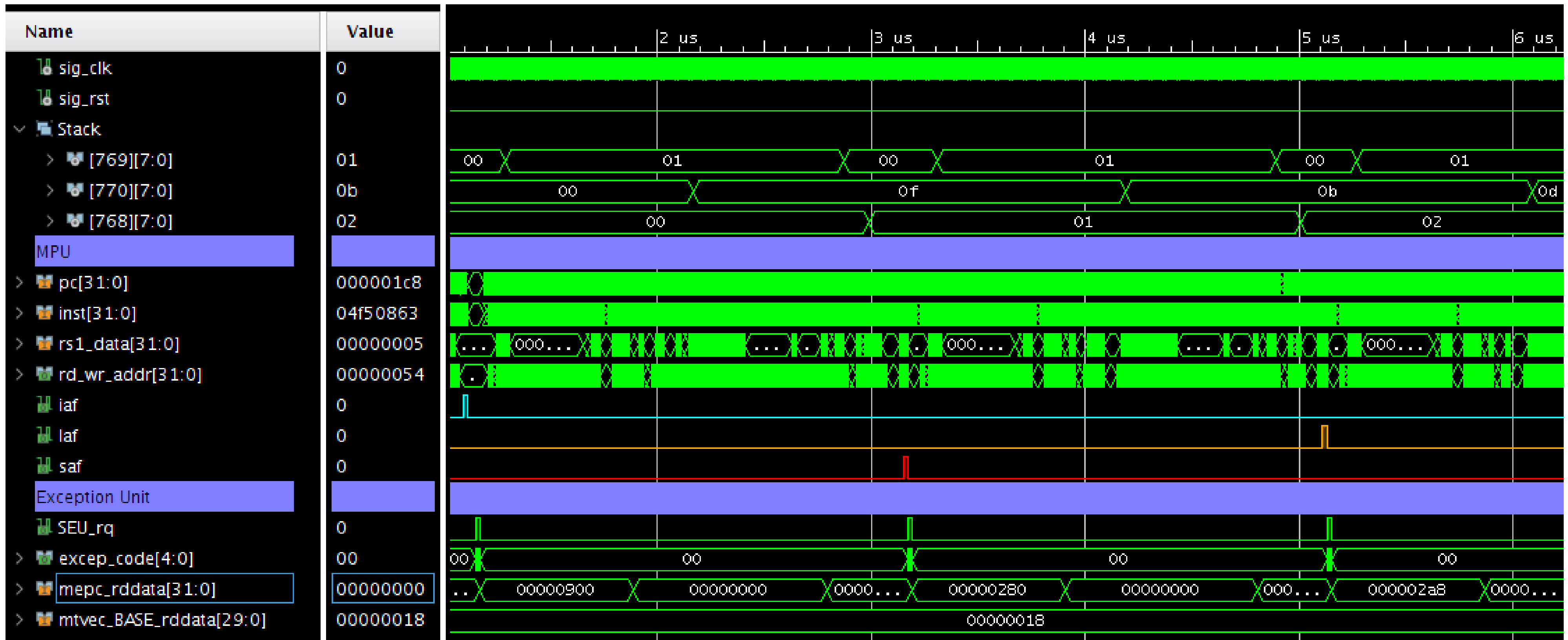

Figure 6.2.: Waveform of the exception handling test

Configuration Change

This test aims to verify the atomic behaviour of an MPU configuration change. This test is crucial to verify the correct interaction between multiple pipeline stages. Especially important are the interactions between the instruction decode (ID) and execute (EX) stage. The control signals for the MPU CSRs originate from the 'CSR_RegisterFile' that is placed in the EX stage of the pipeline. A description of this design can be found in Section 4.1. The MPU registers, however, are as discussed in Section 5.3, instantiated directly as part of the MPU that is placed in the ID stage of the pipeline. In order to not miss any potential illegal access – e.g., after a configuration change – it is necessary that all operations on the CSRs are atomic. Atomic in this context means an uninterruptible operation that only takes one clock cycle to complete.

Figure 6.3 shows the resulting waveform of the test to verify the behaviour described above. For this test, the kernel and exception handler setup of the previous tests was reused. The MPU was configured to allow stores to a region on the stack (512). The *main()* function then calls a subroutine that successfully writes $0x7$ to this region. This time is marked by the first blue cursor in Figure 6.3. The next blue cursor marks the time where the MPU configuration is changed to disallow stores to the previously allowed stack region. This operation takes exactly one clock cycle. On the next rising edge of the clock, a write to the aforementioned stack region is performed. This results in a store access fault (red *saf* signal in Figure 6.3). The test is, therefore passed.

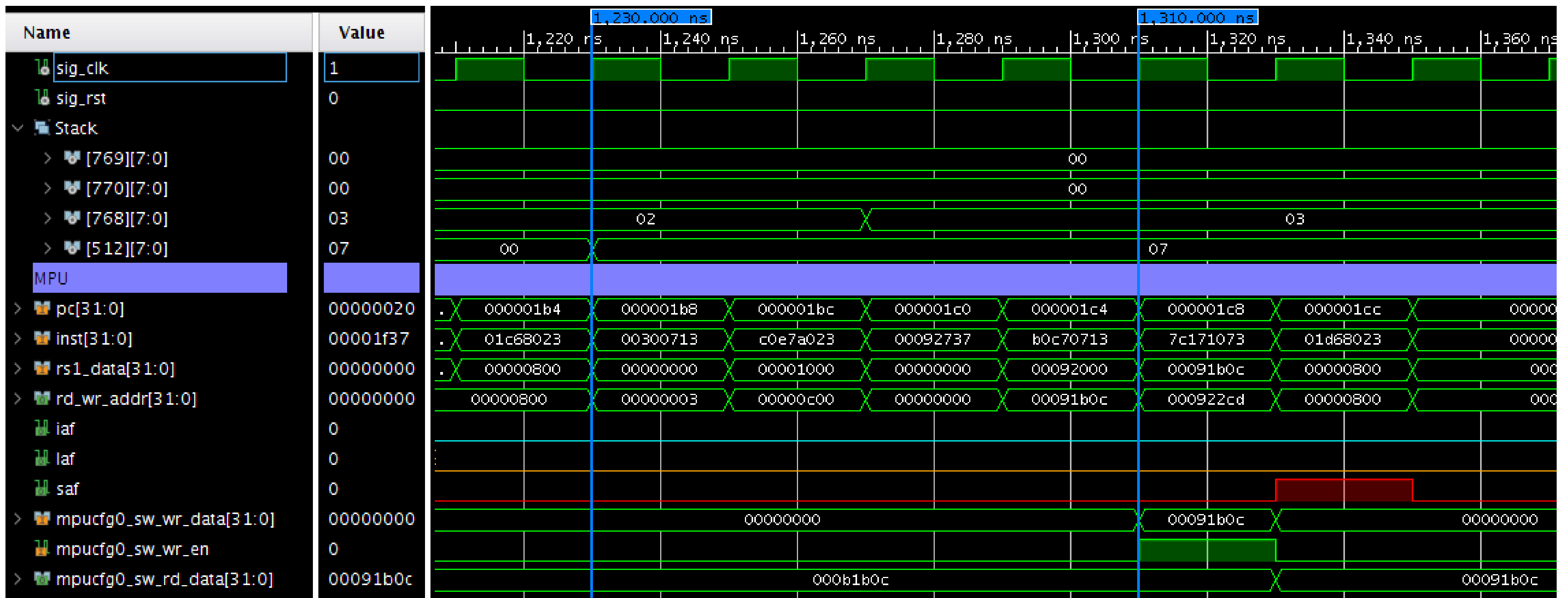

Figure 6.3.: Waveform of the configuration change test

6.2. Resource Requirements

In this section, the results from the design synthesis of the MPU are discussed. Those results are then evaluated in comparison to the core of the 'RiVal' SoC. The MPU was generated with a varying number of supported regions (4, 8, 16, and 32 regions) to see the impact of additional regions.
The synthesis was done with Synopsis Design Compiler (DC), which used the STARLIB_10 technology library (130 μm^2 technology) for cell modelling. To have comparable design reports the synthesis constraints of the RiVal SoC were used for the MPU synthesis as well.

Table 6.1 shows the timing report of the MPU for different configurations. This report is the most important one, as it shows the critical path length of the clock signal. The critical path is the slowest logic path between any two registers, and therefore it is the limiting factor when increasing the clock speed.
It can be observed that an increase in regions does add to the critical path length, but only in a negligible amount. Similarly, the critical path slack – i.e., the greatest difference between the expected and the actual clock arrival time at a gate – increases as well. The slack should be 0. However, since the design is unoptimised and the timing report is generated from a static worst-case timing analysis (i.e., the report is independent of the actual signals which are active when the processor is running), one should not worry too much about reducing negative values [Loc16]. The total negative slack is the sum of all negative critical path slacks. Both total negative slack and critical path slack have the same root cause that could be fixed. Details about the possible fix can be found in Chapter 7.

Timing path in ns	MPU regions			
	4	8	16	32
Critical Path Length	16.978	20.114	21.35	22.461
Critical Path Slack	−17.578	−20.714	−21.95	−23.061
Critical Path Period	23.8	23.8	23.8	23.8
Total Negative Slack	−44.292	−50.21	−54.059	−57.522

Table 6.1.: Timing report of MPU

In Table 6.2 and Table 6.3 the different cell counts and corresponding area can be seen respectively. As expected, the number of hierarchical ports is quite large. This number can be explained because of the of how MPU design is generated. Every gate, function, etc. is placed in a separate source file with the respective in- and out-ports during the ToD transformation. The way the ToD transformation is designed, this step not only leads to a large number of files but also to a great number of ports.
As seen in Table 6.2, there are various cell types. These types are:

- *Hierarchical Cells* are the resulting cells when looking at the design from top-down. Each cell corresponds to a module which is then further divided up into other modules. The result is a hierarchical structure that can be represented as a tree.
- *Leaf Cells* are the lowest level of the hierarchy tree. These cells are all the registers the clock tree is connected to. In a flat design (i.e., a design where all modules are at the same level) every cell is a leaf cell. They perform simple logical operations and consist only of a few transistors. Leaf cells can be either of the four types:
  - *Combinational Cells* are cells whose output depend on the current input. They do not hold any information (i.e., they do not have memory), and the output is time-independent. Examples for such cells are multiplexers, decoders, adders, etc.
  - *Sequential Cells* are cells whose output depend on the current and past input. Therefore, these cells hold information and the output is time depended. An example of such cells is a flip-flop.
  - *Buffer* and *Inverter Cells* are cells that are inserted into the data path. Such cells are required to keep the design synchronised. Otherwise, the delay caused by the clock signal propagation will cause asynchronous behaviour.

The area values seen in Table 6.3 and Table 6.5 are the corresponding area equivalent of the cell type. Their size depends on the technology library used.

Count	MPU regions			
	4	8	16	32
Hierarchical Cell	2102	4137	8207	16 347
Hierarchical Port	159 356	316 290	630 158	1 257 894
Leaf Cell	37 843	76 401	159 616	215 231
Buf/Inv Cell	11 017	23 226	53 904	15 473
Buffer Cell	2540	5392	10 805	1719
Inverter Cell	8477	17 834	43 099	13 754
Combinational Cell	37 619	76 017	158 912	213 887
Sequential Cell	224	384	704	1344

Table 6.2.: Cell report of MPU

Area	MPU regions			
	4	8	16	32
Combinational	514 739.192	1 125 399.996	2 400 406.394	4 869 745.685
Noncombinational	10 752	18 635.2	34 713.599	66 782.398
Buf/Inv	89 379.201	208 464.001	476 649.604	857 969.275
Total Buffer	41 510.4	91 275.199	186 436.798	335 586.235
Total Inverter	47 868.801	117 188.802	290 212.806	522 383.04
Cell	52 491.197	1 144 035.195	2 435 119.993	4 936 528.083
Design	52 491.197	1 144 035.195	2 435 119.993	4 936 528.083

Table 6.3.: Area report of MPU

It can be observed that both cell count and therefore, the area increases linearly the more MPU regions are introduced. The linear increase is also expected, as the more regions have to be supported the more access control blocks have to be generated. Details about the access control blocks implemented in the ToD can be found in Section 5.2.

Table 6.4 and Table 6.5 show the same metrics as before, but this time from the 'RiVal' core with and without an included 32-region MPU.

Count	RiVal	RiVal w/ 32 MPU	Difference
Hierarchical Cell	1928	18 275	+847.9 %
Hierarchical Port	61 981	1 319 875	+2029 %
Leaf Cell	25 103	240 334	+857.4 %
Buf/Inv Cell	8965	24 438	+172.6 %
Buffer Cell	4185	5904	+41.08 %
Inverter Cell	4780	18 534	+287.7 %
Combinational Cell	20 889	234 776	+1023.91 %
Sequential Cell	4209	5553	+31.93 %

Table 6.4.: Cell report of 'RiVal' w/wo MPU

Area	RiVal	RiVal w/ 32 MPU	Difference
Combinational	205 419.201	5 075 164.886	+2371 %
Noncombinational	154 388.797	221 171.195	+43.26 %
Buf/Inv	59 344.001	917 313.276	+1445.76 %
Total Buffer	34 828.8	370 415.035	+963.52 %
Total Inverter	24 515.201	546 898.241	+2130.84 %
Macro/Black Box	803 426.801	803 426.801	0 %
Cell	1 163 234.799	6 099 762.882	+424.4 %
Design	1 163 234.799	6 099 762.882	+424.4 %

Table 6.5.: Area report of 'RiVal' w/wo MPU

When comparing the two variants (with and without MPU), one should not jump to conclusions too early. The increase of 424.4 % in cell area is drastic indeed, however, the design of the 'RiVal' was optimised before and during synthesis, in contrast to the MPU design and synthesis which is unoptimised. Moreover, the 'RiVal' was never synthesised with an integrated MPU. The results presented in Table 6.4 and Table 6.5 were obtained by adding the values from the corresponding MPU table together. It can be assumed that in case of a combined synthesis, the area increase would not be as drastic. This shortcoming

in comparability had to be accepted, as no more time or resources could be spent on the implementation of this work. Further analysis and optimisation are therefore required, which will be left as future work.

6.3. MPU Application Evaluation

This section describes an approach to use the MPU for bare-metal software written in C to enforce data protection and isolation at runtime.

6.3.1. Application Architecture

As described in Section 3.1, it is assumed that the application is comprised of multiple tasks. Each task has its separate MPU configuration, which is stored in a read-only memory section. This section could either be in ROM or in read- and writeable memory marked as read-only through an MPU protection region. As described in Section 3.3, an MPU configuration consists of multiple protection regions.
To describe these regions, the C structure seen in Listing 6.1 can be used. In this structure, four regions are packed together to reduce the memory footprint.

```
1 typedef uint32_t MPU_addr_t;
2 typedef struct MPU_region
3 {
4     MPU_cfg_t cfg;
5     uint32_t num_of_addresses;
6     MPU_addr_t addr[];
7 } MPU_region_t;
```

Listing 6.1.: MPU region structure

This region structure contains a sub-structure for the corresponding configuration entries. As seen in Listing 6.2, the configuration structure is a union of four configuration entries. Using a union saves space, allows convenient access

to individual entries, and makes it easier to write a full configuration register with a single operation.

```c
typedef union MPU_cfg
{
    uint32_t cfg;
    struct
    {
        uint8_t cfg0;
        uint8_t cfg1;
        uint8_t cfg2;
        uint8_t cfg3;
    };
} MPU_cfg_t;
```

Listing 6.2.: MPU configuration structure

The address array stores the values of the MPU address registers. Said array is not fixed in size to reduce the memory overhead in case there are not a multiple of 4 regions available in total. For example, in case there are only 7 regions available, one region structure would contain 4 addresses and another other only 3. This necessitates the inclusion of a variable in the region structure to store the size of the address array. Nevertheless, it should be clear that the address array shall not be larger than 4. Using these two structures, a total of 24 bytes of memory are required to store the configuration for every 4 MPU regions. Each task needs its configuration, so the overall memory footprint increases with every task. For example, given a fully filled region structure and two tasks, 48 bytes are required.

Per default, the MPU should be programmed that all data in the random access memory (RAM) area is read-only. Only the stack of a task and the memory-mapped IO regions it needs to access are writeable. It should be kept in mind that the size of the task stack needs to be a multiple of power-of-two bytes. Since, as described in Section 3.3, only a NAPOT range can be used to specify a range larger than 4 bytes for read/writeable regions. Furthermore, the executable region of a task needs to be specified as well. This also includes the interrupt service routines. By assuming that a dedicated exception handler handles all

interrupts in the kernel, the executable region of a task could be simplified. Upon kernel entry, the MPU will be disabled and only enabled when the kernel is exited again. This approach requires that at least the kernel entry and exit are in an executable region. This region should have its own MPU configuration entry that is locked into place to circumvent unintentional modification of the kernel region configuration. Arguably, the whole kernel region could be made executable and readable for all tasks. However, this would lead the idea to have a single trusted central authority ad absurdum. Since this would allow every task to access kernel functions without the need for a prior context switch, this approach is strongly discouraged. Overall, the recommended approach requires 4 MPU regions for a minimal working example (read-only RAM, stack, the executable region of the task, and kernel entry/exit).

The MPU configuration change from one task to another has to happen during a period when the MPU is disabled. The obvious choice for this is during the task context switch in the kernel. During this context switch, the tasks register values and stack are loaded, so it only needs to be extended to load the configuration of the MPU for the new task. If all MPU configurations are in consecutive memory, such is the case with an array, the task ID can be used to get the correct configuration. An example of such a function can be seen in Listing 6.3.

For the function shown in Listing 6.3 it is assumed that the MPU configurations are stored in a specific memory segment. The start of this memory segment is exposed by a linker variable *MPU_cfgs* during the linking process. Furthermore, another variable is required. The variable *MPU_cfgs_size*, initialised during the creation of all configurations, stores the size of the configuration array.

In Listing 6.3 specific functions to write to the MPU address and configuration registers are used. The function to write to the address registers is presented in Listing 6.4. To write to the configuration registers a similar function could be used.

6.3.2. Application Example

This section introduces the application to test the performance penalty when using the MPU with the MPU usage example presented in Section 6.3.1.

```
extern MPU_region_t** MPU_cfgs; // region exposed by linker
extern uint32_t MPU_cfgs_size; // defined somewhere else

void MPU_load_cfg(const uint32_t ID)
{
    if (MPU_cfgs && MPU_cfgs[ID])
    {
        MPU_region_t* new_cfg = MPU_cfgs[ID];
        uint32_t address_counter = 0;
        for (uint32_t i = 0; i < MPU_cfgs_size; i++)
        {
            /* write MPU configuration registers
            MPU_write_cfg_reg(uint32_t number, uint32_t value) */
            MPU_write_cfg_reg(i, new_cfg[i].cfg.cfg);
            for (uint32_t j = 0; j < new_cfg[i].num_of_addresses; j++)
            {
                /* write MPU address registers
                MPU_write_addr_reg(uint32_t number, uint32_t value) */
                MPU_write_addr_reg(address_counter++, new_cfg[i].addr[j]);
            }
        }
    }
    return;
}
```

Listing 6.3.: MPU configuration load function

To test the system performance and increase in context switch time, a modified version of the Dhrystone benchmark was used. Dhrystone is a synthetic benchmark program developed by Weicker [Wei84] in 1984 to test the performance of general-purpose CPUs. This benchmark was widely adopted and modified to run on various systems and architectures. Because it is lightweight and offers excellent portability, it was chosen to test the application overhead when using the proposed application usage example.

By default, the Dhrystone benchmark is comprised of 8 routines out of which 7 are called directly by the main routine. The benchmark was modified to

```c
#define MPU_ADDR0 0x7C9
#define MPU_ADDR1 0x7CA
/* ... */

void MPU_write_addr_reg(const uint32_t number, const uint32_t value)
{
    switch (number)
    {
        case 0:
        __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR0), "r"(value));
        break;
        case 1:
        __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR1), "r"(value));
        break;
        /* ... */
        default:
        break;
    }
    return;
}
```

Listing 6.4.: MPU address write function

perform a context switch with the help of the *ecall* instruction right before each call of those routines. The exception handler then loads the MPU with the to the routine associated configuration. Doing so simulates the task switching described in Section 6.3.1. Therefore, the overhead imposed when using the MPU can be measured for a varying number of protection regions. It should be noted that for the benchmarking dummy values were used to load into the registers, and the MPU was kept deactivated. Doing so further reduced the complexity of the test without impacting the benchmark results.
The listings of the source code used can be found in Appendix B.

6.3.3. Results

The application example introduced in Section 6.3.2 was run with the same setup as the behavioural tests described in Section 6.1. To measure the impact of the MPU on the system performance, 6 Dhrystone variations were used. Those variations were:

1. the vanilla Dhrystone benchmark without any context switching.
2. the Dhrystone benchmark with a context switch before each Dhrystone routine call.
3. the context switch benchmark with a complete MPU configuration change on each context switch for an MPU supporting 4, 8, 16, and 32 regions.

Each benchmark variation was run 10 000 times at a clock rate of 50 MHz.

The resulting runtime and benchmarking results can be seen in Table 6.6.

Test	Runs	Clk	Runtime	DPS	DMIPS	DMIPS/MHz
	#	MHz	µs			
Dhrystone	10 000	50	166 200.04	60 168.46	34.24	0.68
Context Switch	10 000	50	257 200.04	38 880.24	22.13	0.44
4 region MPU	10 000	50	549 200.10	18 208.30	10.36	0.21
8 region MPU	10 000	50	736 400.10	13 579.56	7.73	0.15
16 region MPU	10 000	50	1 115 600.10	8963.79	5.10	0.10
32 region MPU	10 000	50	1 955 600.10	5113.52	2.91	0.06

Table 6.6.: Dhrystone Benchmark Results

Given the runtime T and the number of runs N, the following formula can be used to calculate the benchmark results in Dhrystones per second (DPS).

$$\text{DPS} = \frac{N}{T}$$

To make this number more comparable, the VAX 11/780 was selected as the reference 1 ‘million of instructions per second’ (MIPS) machine by the industry.

This is necessary since the literal comparison of DPS and therefore MIPS is not meaningful when comparing, for example, RISC and CISC based processors [11]. Said processor achieves 1757 DPS, so the resulting DMIPS are calculated using:

$$\text{DMIPS} = \frac{\text{DPS}}{1757}$$

The DMIPS/MHz rating normalises the result further, by including the clock rate a processor is run at. Given a clock rate *clk* in MHz the DMIPS/MHz is calculated with the following formula:

$$\frac{\text{DMIPS}}{\text{MHz}} = \frac{\text{DMIPS}}{clk}$$

The most important metric to compare for this thesis, however, remains the total runtime. This metric allows measuring the impact on context switch times when using an MPU. The runtimes from the Dhrystone benchmark are depicted in Figure 6.4.

As expected, the DMIPS go down when a context switch is introduced in the benchmark. This reduction is because the inclusion of a context switch increases the total runtime. A depiction of this increase can be seen in Figure 6.4. In this figure, it can be seen that the runtime is increased by 55 %. When an MPU reconfiguration is part of every context switch the runtime is more than doubled once more. Naturally, the more regions there are to configure, the longer the reconfiguration takes. A closer look at Figure 6.4 shows a nearly linear relationship between the number of regions and the required reconfiguration time, which is expected behaviour. Overall, one cannot help to notice that the reconfiguration consumes the most time. So much in fact, that the total runtime is increased by factor 11.5 when using a 32 region MPU compared to the vanilla benchmark runtime.
However, the presented results should be taken with a grain of salt. The test presented in Section 6.3.2 could be interpreted as the worst-case application example. One reason for this is because the sub-routines used to emulate task behaviour are often no longer than a few lines of code. Moreover, since a context switch with subsequent MPU reconfiguration is forced before every call,

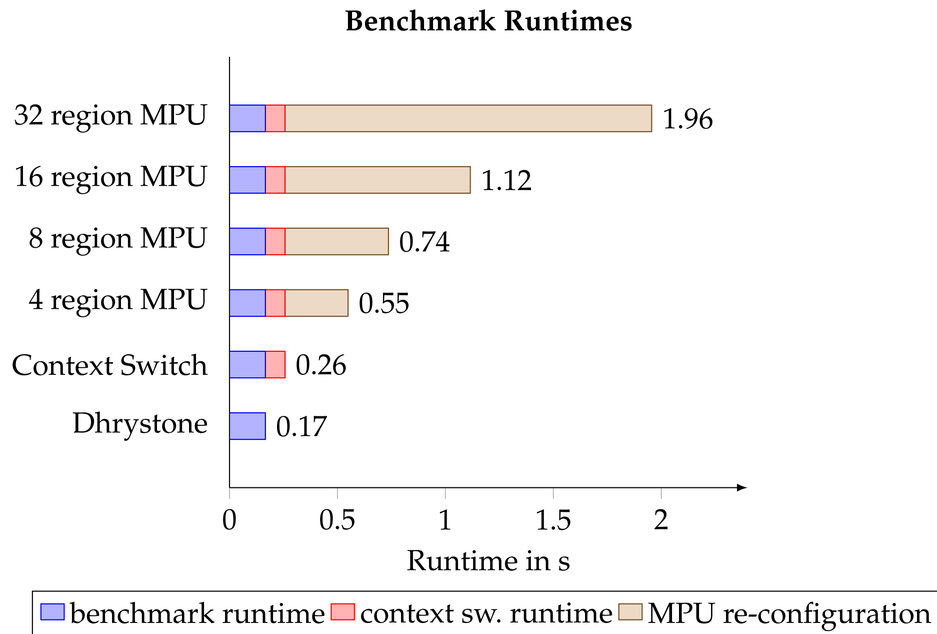

Figure 6.4.: Dhrystone Benchmark runtimes

it is obvious why so much time is spent on not running the task. Doing an excessive amount of context switches in such a short period is something that not usually observed in real-world applications.
Nevertheless, the runtime results clearly show that the number of MPU re-configurations should be kept at a minimum. This could be achieved by only changing the MPU entries that need to be changed instead of reloading a possible unchanged configuration. For example, in case a kernel service is requested by a task, the MPU should not be re-configured every time.

To truly determine the performance impact of the MPU, a more thorough study is required. More specifically, the performance degradation of a 'real-world' application needs to be examined. Said evaluation could not be done as part of this thesis for multiple reasons. First and foremost, the deadline for hand-in could not be moved any further. And second, with the limited resources of the simulated SoC, no actual application could be run. The resource constraint with the greatest impact was the memory with only 4 kB of RAM and ROM.

This amount of memory should be sufficient for an application; however, the Harvard architecture of the CPU makes it impossible to read data from the ROM. Therefore, all data needed for computation has to be loaded in the RAM first. This also includes the MPU configurations. As mentioned in Section 6.3.1, 24 bytes of memory are required to store the configuration of 4 MPU regions per task. Given these constraints, this means that for 8 tasks and a 32 region MPU a total of 1536 bytes of memory are required. Those constraints make it very challenging to develop an application. Further performance analysis will be therefore left as future work.

7. Summary and Future Work

In the previous chapters, it was shown how an MPU for a RISC-V architecture can be generated based on the model-driven architecture (MDA) principle for hardware generation.
The described approach is a novel technique to generate an MPU that was left unexplored by both academia and industry. The described MPU was not only shown to be fully functional but also highly adaptable for various use cases. Similar specifications and design ideas of the presented MPU and the MMU described in the RISC-V ISA specification, make it possible to transform the MPU into a full-fledged RISC-V compliant MMU with minimal additional work.
The described approach, however, is not fully mature and has room for improvement.

First and foremost, the MPU design needs to be reworked to be 'clock clean', which means that the total negative slack in the clock path is no more than 0. The slack of the current design can be seen in Section 6.2. Fortunately, most of the slack has the same root cause that could be fixed. According to the longest path analysis during synthesis, the culprit is the addition of the register data and the address offset in the decoder structure. A depiction of this structure can be seen in Figure 5.5. The addition itself cannot be removed; however, the impact on the longest path can be mitigated with the introduction of a stall signal. This stall signal will be set to HIGH on every load and store instruction until the addition operation is complete. By connecting this signal to the other pipeline stall signals, there will be enough time for the MPU to complete necessary access control checks.
Second, the introduction of another stall signal to the pipeline will have a negative performance impact. This impact should be evaluated with a variety of applications. This application study is also required to measure the real impact on context switch times, as discussed in Section 6.3.3. Many applications

that could be used for performance evaluations require the MPU to be included in a more powerful and bigger CPU. Especially the memory is a limiting factor, as mentioned in Section 6.3.3.
Third, the resource requirements need to be re-evaluated after possible design optimisations. The synthesis should also be done with optimisations enabled. At the time of writing this thesis, no optimisations were activated, as described in Section 6.2.
Finally, as described in Section 3.1, it was assumed that there are multiple privilege modes available. Currently, only one privilege mode is available. The other modes are yet to be implemented.

Bibliography

[11] *Dhrystone Benchmarking for ARM Cortex Processors*. AN273. 20th July 2011. URL: https://developer.arm.com/documentation/dai0273/a/ (visited on 16/09/2020).

[17] *ARMv8-M Fault handling and detection*. en. 100690. Version 2.0. 28th Feb. 2017. URL: https://developer.arm.com/docs/100691/0200 (visited on 26/03/2020).

[19a] *ARMv8-M Memory Protection Unit*. en. Version 2.1. 9th Apr. 2019. URL: https://static.docs.arm.com/100699/0201/armv8m_memory_protection_unit_100699_0201_en.pdf?_ga=2.214628055.1991967102.1591687898-1399787952.1585051259.

[19b] *Microcontroller pocket guide*. Munich: Infineon Technologies AG, 2019. URL: https://www.infineon.com/dgdl/Infineon-Microcontroller_Pocket_Guide__2019-ProductSelectionGuide-v01_01-EN.pdf?fileId=5546d46259d9a4bf015a282ef33c3e46 (visited on 17/08/2020).

[Bac+12] Jonathan Bachrach et al. 'Chisel: Constructing hardware in a Scala embedded language'. In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584.

[CP13] Ron Collett and Dorian Pyle. *What happens when chip-design complexity outpaces development productivity?* McKinsey & Company. 2013. URL: https://www.mckinsey.com/~/media/McKinsey/dotcom/client_service/Semiconductors/Issue%203%20Autumn%202013/PDFs/4_ChipDesign.ashx (visited on 08/01/2020).

[Eck+14] Wolfgang Ecker et al. 'The Metamodeling Approach to System Level Synthesis'. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. Dresden, Germany: European Design and Automation Association, 2014. ISBN: 9783981537024.

[ES16] Wolfgang Ecker and Johannes Schreiner. 'Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient hardware generators'. In: *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (26th Sept. 2016). Tallinn, Estonia: IEEE, Sept. 2016, pp. 1–6. ISBN: 978-1-5090-3561-8. DOI: 10.1109/VLSI-SoC.2016.7753576.

[FDM19] Muhammad Faisal, Erik Dilger and Sergio Montenegro. 'A Unified Approach for Memory Protection in a Bare-Metal and a Real Time Operating System'. In: *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*. PCI '19. Nicosia, Cyprus: Association for Computing Machinery, 2019, pp. 149–152. ISBN: 9781450372923. DOI: 10.1145/3368640.3368667.

[HRK12] Anton Hattendorf, Andreas Raabe and Alois Knoll. 'Shared memory protection for spatial separation in multicore architectures'. In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 2012, pp. 299–302. ISBN: 978-1-4673-2684-1. DOI: 10.1109/SIES.2012.6356601.

[ISO18] ISO/TC 22/SC 32. *ISO 26262:2018 - Road Vehicles–Functional Safety*. en. Standard. Geneva, CH, 2018.

[Jan+19] Hyeonguk Jang et al. 'MMNoC: Embedding Memory Management Units into Network-on-Chip for Lightweight Embedded Systems'. In: *IEEE Access* 7 (2019), pp. 80011–80019. DOI: 10.1109/access.2019.2923219.

[KM05] Martin Kempa and Zoltán Adám Mann. 'Model Driven Architecture'. In: *Informatik-Spektrum* 28.4 (2005), pp. 298–302. DOI: 10.1007/s00287-005-0505-2.

[Loc16] Derek Lockhart. *RTL-to-Gates Synthesis using Synopsys Design Compiler. ECE5745 Tutorial 2*. Version 606ee8a. 30th Jan. 2016. URL: https://www.csl.cornell.edu/courses/ece5745/handouts/ece5745-tut2-dc.pdf (visited on 01/10/2020).

[Lop14] Lanfranco Lopriore. 'Hardware support for memory protection in sensor nodes'. In: *Microprocessors and Microsystems* 38.3 (May 2014), pp. 226–232. DOI: 10.1016/j.micpro.2014.01.004.

[Lop16] Lanfranco Lopriore. 'Memory protection in embedded systems'. In: *Journal of Systems Architecture* 63 (Feb. 2016), pp. 61–69. DOI: 10.1016/j.sysarc.2016.01.006.

[Mal+15] Moshe Malka et al. 'rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 355–368. ISBN: 9781450328357. DOI: 10.1145/2694344.2694355.

[MB19] Maja Malenko and Marcel Baunach. 'Device Driver and System Call Isolation in Embedded Devices'. English. In: *Proceedings - Euromicro Conference on Digital System Design, DSD 2019*. Ed. by Nikos Konofaos and Paris Kitsos. Proceedings - Euromicro Conference on Digital System Design, DSD 2019. United States: Institute of Electrical and Electronics Engineers, Aug. 2019, pp. 283–290. DOI: 10.1109/DSD.2019.00049.

[MCF03] Stephen J. Mellor, Anthony N. Clark and Takao Futagami. 'Guest Editors' Introduction: Model-Driven Development'. In: *IEEE Softw.* 20.5 (Sept. 2003), pp. 14–18. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231145.

[Pau18] Sebastian Paulus. 'Implementing IO Pads in HW/SW using the Model-Driven Architecture Vision'. Bachelor's Thesis. Technical University of Munich, 2018.

[PBB18] Francesco Paci, Davide Brunelli and Luca Benini. 'Lightweight IO Virtualization on MPU Enabled Microcontrollers'. In: *SIGBED Rev.* 15.1 (Mar. 2018), pp. 50–56. DOI: 10.1145/3199610.3199617.

[PGS11] Joel Porquet, Alain Greiner and Christian Schwarz. 'NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs'. In: *2011 Design, Automation & Test in Europe* (2011), pp. 1–4. ISSN: 1558-1101. DOI: 10.1109/DATE.2011.5763291.

[PP19] Runyu Pan and Gabriel Parmer. 'MxU: Towards Predictable, Flexible, and Efficient Memory Access Control for the Secure IoT'. In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). ISSN: 1539-9087. DOI: 10.1145/3358224.

[PW17] David Patterson and Andrew Waterman. *The RISC-V Reader. An Open Architecture Atlas. The RISC-V Reader: An Open Architecture Atlas*. First edition. Strawberry Canyon LLC, 7th Nov. 2017. ISBN: 9780999249116.

[Riv18] J. Germán Rivera. 'Hardware-Based Data Protection/Isolation at Runtime in Ada Code for Microcontrollers'. In: *Ada Lett.* 37.2 (June 2018), pp. 43–50. ISSN: 1094-3641. DOI: 10.1145/3232693.3232705.

[SB20] Tobias Scheipel and Marcel Baunach. 'papagenoX: Generation of Electronics and Logic for Embedded Systems from Application Software'. English. In: *Proceedings of the 9th International Conference on Sensor Networks*. INSTICC. SCITEPRESS - Science and Technology Publications, Feb. 2020, pp. 136–141. ISBN: 978-989-758-403-9. DOI: 10.5220/0009159701360141.

[Sch16] Johannes Schreiner. 'Automated generation of pipelined RISC CPUs following the Model-driven Architecture principle'. MA thesis. Technical University of Munich, 2016.

[Sha+16] Farid Shamani et al. 'Integration issues of a run-time configurable memory management unit to a RISC processor on FPGA'. In: *Microprocessors and Microsystems* 49 (Mar. 2016), pp. 179–191. DOI: 10.1016/j.micpro.2016.12.001.

[SLM13] Oliver Stecklina, Peter Langendoerfer and Hannes Menzel. 'Design of a tailor-made memory protection unit for low power microcontrollers'. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2013, pp. 225–231. ISBN: 978-1-4799-0658-1. DOI: 10.1109/SIES.2013.6601495.

[Sti12] Michael Stilkerich. 'Memory protection at option: application-tailored memory safety in safety-critical embedded systems'. PhD thesis. University of Erlangen-Nuremberg, 2012. URL: http://www4.cs.fau.de/~mike/pdf/MichaelStilkerichDissertation.pdf (visited on 17/08/2020).

[WA19a] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20191213. RISC-V Foundation. RISC-V Foundation, 13th Dec. 2019. URL: https://riscv.org/specifications/ (visited on 02/04/2020).

[WA19b] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20190608-Priv-MSU-Ratified. RISC-V Foundation. RISC-V Foundation, 8th June 2019. URL: https://riscv.org/specifications/privileged-isa (visited on 01/07/2019).

[Wei84] Reinhold P Weicker. 'Dhrystone: a synthetic systems programming benchmark'. In: *Communications of the ACM* 27.10 (1984), pp. 1013–1030.

[YN14] Shimpei Yamada and Yukikazu Nakamoto. 'Protection Mechanism in Privileged Memory Space for Embedded Systems, Real-Time OS'. In: *2014 IEEE 34th International Conference on Distributed Computing Systems Workshops*. IEEE, June 2014. DOI: 10.1109/icdcsw.2014.17.

[Zap18] Matteo Zappia. 'Modeling of RISC-V Exceptions for Hardware Code Generation'. MA thesis. Technical University of Munich, 28th Nov. 2018.

Appendix A.

Behavioural Test Source Code

A.1. Kernel and Exception Handler

```
1  #include "common.h"
2  #include "kernel.h"
3
4  volatile uint32_t* enabled_interrupts;
5
6  extern void kernel_entry(void);
7
8  #define MTVEC_ADDR 0x305U
9
10 void exception_handler(uint32_t cause, uint32_t pc, uint32_t value)
11 {
12   switch (cause)
13   {
14   case 1U: // iaf
15     *((uint32_t*)0xC08) = 0xF;
16     break;
17   case 5U: // laf
18     *((uint32_t*)0xC08) = 0xD;
19     break;
20   case 7U: // saf
21     *((uint32_t*)0xC08) = 0xB;
22     break;
23   case 11U: // mecall
24     *((uint32_t*)0xC08) = 0x9;
25   default:
```

```
26      break;
27    }
28    return;
29  }
30
31  void setup_exceptions(void)
32  {
33    enabled_interrupts = (uint32_t*)0xC50;
34    CSR_WRITE((uint16_t*)MTVEC_ADDR, ((uint32_t)&kernel_entry) << 2U);
35  }
```

A.1.: kernel.c for behavioural tests

```
1   #define cause t0
2   #define handler t2
3
4   .extern exception_handler
5
6   .global kernel_entry
7   kernel_entry:
8     li t6, 1
9     li t5, 0xC04
10    sb t6, 0(t5)
11    /* save current context */
12    addi sp,sp,-68
13    sw sp,64(sp)
14    sw ra,60(sp)
15    sw t0,56(sp)
16    sw t1,52(sp)
17    sw t2,48(sp)
18    sw a0,44(sp)
19    sw a1,40(sp)
20    sw a2,36(sp)
21    sw a3,32(sp)
22    sw a4,28(sp)
23    sw a5,24(sp)
24    sw a6,20(sp)
25    sw a7,16(sp)
26    sw t3,12(sp)
27    sw t4,8(sp)
```

```
28    sw t5,4(sp)
29    sw t6,0(sp)
30  get_cause:
31    csrrw cause, mcause, zero
32    li t1, 0x7FFFFFFF
33    bleu cause, t1, exc_handler
34    j dispatch
35  exc_handler:
36    la handler, exception_handler
37    /* prepare for function call */
38    addi sp, sp, -4
39    sw ra, 0(sp)
40    mv a0, cause
41    csrrw a1, mepc, zero /* pass offending PC as function parameter */
42    mv a2, a7
43    jalr handler /* call the handler */
44    lw ra, 0(sp)
45    addi sp, sp, 4
46  dispatch:
47    /* restore context */
48    lw sp,64(sp)
49    lw ra,60(sp)
50    csrw mepc, ra /* setup address to return to */
51    lw t0,56(sp)
52    lw t1,52(sp)
53    lw t2,48(sp)
54    lw a0,44(sp)
55    lw a1,40(sp)
56    lw a2,36(sp)
57    lw a3,32(sp)
58    lw a4,28(sp)
59    lw a5,24(sp)
60    lw a6,20(sp)
61    lw a7,16(sp)
62    lw t3,12(sp)
63    lw t4,8(sp)
64    lw t5,4(sp)
65    lw t6,0(sp)
66    addi sp,sp,68
67    li t6, 0
68    li t5, 0xC04
```

```
69   sb t6, 0(t5)
70   mret
```

A.2.: kernel.S

A.2. Exception Handling Test

```
1  #include "common.h"
2  #include "kernel.h"
3
4  #define STATUS_ADDR 0xFC0
5  #define CTRL_ADDR 0x7C0
6  #define CFG0_ADDR 0x7C1
7  #define ADDR0_ADDR 0x7C3
8  #define ADDR1_ADDR 0x7C4
9  #define ADDR2_ADDR 0x7C5
10 #define MSEE_ADDR 0x307
11
12 void exec_fault(void);
13 void store_fault_wrong_size(const uint16_t value);
14 void store_fault_wrong_byte(const uint8_t value);
15 void store_fault_store_not_allowed(const uint8_t value);
16 uint16_t load_fault_wrong_size(void);
17 uint8_t load_fault_wrong_byte(void);
18 void configure_mpu(void);
19
20 // put function outside of allowed IM region
21 void __attribute__((section(".locked"))) exec_fault(void)
22 {
23   store_fault_wrong_size(0xFF);
24   return;
25 }
26
27 void store_fault_wrong_size(const uint16_t value)
28 {
29   __asm__ volatile("sh %0, 0(%1)" : : "r"(value), "r"(0x801));
30   return;
31 }
```

```
32
33 void store_fault_store_not_allowed(const uint8_t value)
34 {
35   __asm__ volatile("sb %0, 0(%1)" : : "r"(value), "r"(0x801));
36   return;
37 }
38
39 uint16_t load_fault_wrong_size(void)
40 {
41   uint16_t value;
42   __asm__ volatile("lhu %0, 0(%1)" : "=r"(value) : "r"(0x801));
43   return value;
44 }
45
46 uint8_t load_fault_wrong_byte(void)
47 {
48   uint8_t value;
49   __asm__ volatile("lbu %0, 0(%1)" : "=r"(value) : "r"(0x800));
50   return value;
51 }
52
53 void configure_mpu(void)
54 {
55   // disable MPU
56   CSR_WRITE((uint16_t*)CTRL_ADDR, 0U);
57   // configure MPU for operation tests
58   // L=0, B=0, A=TOR, X=1, W=0, R=0  (IM)
59   uint8_t cfg0 = 0b00001100;
60   // L=0, B=0, A=NAPOT, X=0, W=1, R=1 (stack region)
61   uint8_t cfg1 = 0b00011011;
62   // L=0, B=0, A=BM, X=0, W=1, R=1 (some input)
63   uint8_t cfg2 = 0b00101001;
64   // TOR range 0x0 - 0x900 (instruction memory)
65   uint32_t addr0 = 0x900;
66   // NAPOT range 0xC00 - 0x1000 (stack region)
67   uint32_t addr1 = (0xC00 + (0x400 >> 1U) - 1) >> 2U;
68   // BM 0x800
69   uint32_t addr2 = 0x800;
70   CSR_WRITE((uint16_t*)CFG0_ADDR,
71             (uint32_t)((cfg2 << 16) | (cfg1 << 8) | cfg0));
72   CSR_WRITE((uint16_t*)ADDR0_ADDR, addr0);
```

```
73   CSR_WRITE((uint16_t*)ADDR1_ADDR, addr1);
74   CSR_WRITE((uint16_t*)ADDR2_ADDR, addr2);
75   // enable mecall, saf, laf, iaf exceptions
76   CSR_WRITE((uint16_t*)MSEE_ADDR, 0x8A2);
77   // enable MPU
78   CSR_WRITE((uint16_t*)CTRL_ADDR, 1U);
79   return;
80 }
81
82 void __attribute__((noreturn, optimize("O0"))) main(void)
83 {
84   setup_exceptions();
85   configure_mpu();
86   uint8_t tmp = 27U;
87   while (1)
88   {
89     exec_fault();
90     *((volatile uint32_t*)0xC00) = 1; // for debugging
91     store_fault_store_not_allowed(tmp);
92     *((volatile uint32_t*)0xC00) = 2; // for debugging
93     tmp = load_fault_wrong_byte();
94     *((volatile uint32_t*)0xC00) = 3; // for debugging
95   }
96 }
```

A.3.: exception_test.c

A.3. Configuration Change Test

```
1 #include "common.h"
2 #include "kernel.h"
3
4 #define STATUS_ADDR 0xFC0
5 #define CTRL_ADDR 0x7C0
6 #define CFG0_ADDR 0x7C1
7 #define ADDR0_ADDR 0x7C3
8 #define ADDR1_ADDR 0x7C4
9 #define ADDR2_ADDR 0x7C5
```

```
10 #define MSEE_ADDR 0x307
11
12 void configure_mpu(void);
13
14 void change_test(void)
15 {
16   register uint8_t value_1 asm("t3") = 0x7;
17   register uint8_t value_2 asm("t4") = 0x8;
18   // for debugging
19   *((volatile uint32_t*)0xC00) = 2;
20   // success
21   __asm__ volatile("sb %0, 0(%1)" : : "r"(value_1), "r"(0x800));
22   // for debugging
23   *((volatile uint32_t*)0xC00) = 3;
24   // change config
25   CSR_WRITE((uint16_t*)CFG0_ADDR, 0b00001001000110110000l100);
26   // fail
27   __asm__ volatile("sb %0, 0(%1)" : : "r"(value_2), "r"(0x800));
28   // for debugging
29   *((volatile uint32_t*)0xC00) = 4;
30   return;
31 }
32
33 void configure_mpu(void)
34 {
35   // disable MPU
36   CSR_WRITE((uint16_t*)CTRL_ADDR, 0U);
37   // configure MPU for operation tests
38   // L=0, B=0, A=TOR, X=1, W=0, R=0  (IM)
39   uint8_t cfg0 = 0b00001100;
40   // L=0, B=0, A=NAPOT, X=0, W=1, R=1 (stack region)
41   uint8_t cfg1 = 0b00011011;
42   // L=0, B=0, A=BM, X=0, W=1, R=1 (some input)
43   uint8_t cfg2 = 0b00001011;
44   // TOR range 0x0 - 0x900 (instruction memory)
45   uint32_t addr0 = 0x900;
46   // NAPOT range 0xC00 - 0x1000 (stack region)
47   uint32_t addr1 = (0xC00 + (0x400 >> 1U) - 1) >> 2U;
48   // BM 0x800
49   uint32_t addr2 = 0x800;
50   CSR_WRITE((uint16_t*)CFG0_ADDR,
```

```
51            (uint32_t)((cfg2 << 16) | (cfg1 << 8) | cfg0));
52   CSR_WRITE((uint16_t*)ADDR0_ADDR, addr0);
53   CSR_WRITE((uint16_t*)ADDR1_ADDR, addr1);
54   CSR_WRITE((uint16_t*)ADDR2_ADDR, addr2);
55   // enable mecall, saf, laf, iaf exceptions
56   CSR_WRITE((uint16_t*)MSEE_ADDR, 0x8A2);
57   // enable MPU
58   CSR_WRITE((uint16_t*)CTRL_ADDR, 1U);
59   return;
60 }
61
62 void __attribute__((noreturn, optimize("O0"))) main(void)
63 {
64   setup_exceptions();
65   configure_mpu();
66   // for debugging
67   *((volatile uint32_t*)0xC00) = 1;
68   change_test();
69   // for debugging
70   *((volatile uint32_t*)0xC00) = 5;
71   while (1)
72     ;
73 }
```

A.4.: change_test.c

Appendix B.

Application Performance Test Source Code

B.1. Dhrystone Source Code

```
#include "common.h"
#include "kernel.h"
#include "dhry.h"
#include "mpu.h"

#define NUMBER_OF_RUNS ((uint32_t)10000U)
#define NUMBER_OF_REGIONS 8
#define NUMBER_OF_CFGS 8

#define NOSTRUCTASSIGN 1

uint32_t dhrystone_benchmark(void);

/* Global Variables: */

Rec_Pointer Ptr_Glob, Next_Ptr_Glob;
int Int_Glob;
Boolean Bool_Glob;
char Ch_1_Glob;
char Ch_2_Glob;
int Arr_1_Glob[20];
int Arr_2_Glob[20][20];

```

```
24 #ifndef REG
25 Boolean Reg = false;
26 #define REG
27 /* REG becomes defined as empty */
28 /* i.e. no register variables   */
29 #else
30 Boolean Reg = true;
31 #endif
32
33 void Proc_1(REG Rec_Pointer);
34 void Proc_2(One_Fifty*);
35 void Proc_3(Rec_Pointer*);
36 void Proc_4();
37 void Proc_5();
38
39 void Proc_6(Enumeration, Enumeration*);
40 void Proc_7(One_Fifty, One_Fifty, One_Fifty*);
41 void Proc_8(Arr_1_Dim, Arr_2_Dim, int, int);
42 Enumeration Func_1(Capital_Letter, Capital_Letter);
43 Boolean Func_2(Str_30, Str_30);
44 char* m_strcpy(char* destination, const char* source);
45 void memcpy(register char*, register char*, register int);
46
47 void* m_malloc_1(int nbytes);
48 void* m_malloc_2(int nbytes);
49
50 void init_string_some(void);
51 void init_string_1(void);
52 void init_string_2(void);
53 void init_string_3(void);
54
55 static char string_some[31];
56 static char string_1[31];
57 static char string_2[31];
58 static char string_3[31];
59
60 Str_30 Str_1_Loc;
61 Str_30 Str_2_Loc;
62
63 void __attribute__((noreturn)) main(void)
64 {
```

```
65  #ifdef CONTEXT_SWITCH
66    setup_exceptions();
67  #ifdef MPU_ACTIVE
68    setup_MPU_regions(NUMBER_OF_REGIONS, NUMBER_OF_CFGS);
69  #endif
70  #endif
71    dhrystone_benchmark();
72    while (1)
73      ;
74    __builtin_unreachable();
75  }
76
77  uint32_t dhrystone_benchmark(void)
78  /*****/
79  /* main program, corresponds to procedures        */
80  /* Main and Proc_0 in the Ada version             */
81  {
82    One_Fifty Int_1_Loc;
83    REG One_Fifty Int_2_Loc;
84    One_Fifty Int_3_Loc;
85    REG char Ch_Index;
86    Enumeration Enum_Loc;
87    REG uint32_t Run_Index;
88
89    /* Initializations */
90    init_string_some();
91    init_string_1();
92    init_string_2();
93    init_string_3();
94    Next_Ptr_Glob = (Rec_Pointer)m_malloc_1(
95        sizeof(Rec_Type)); // change Malloc to malloc
96    Ptr_Glob = (Rec_Pointer)m_malloc_2(
97        sizeof(Rec_Type)); // change Malloc to malloc
98
99    Ptr_Glob->Ptr_Comp = Next_Ptr_Glob;
100   Ptr_Glob->Discr = Ident_1;
101   Ptr_Glob->variant.var_1.Enum_Comp = Ident_3;
102   Ptr_Glob->variant.var_1.Int_Comp = 40;
103   m_strcpy(Ptr_Glob->variant.var_1.Str_Comp, string_some);
104   m_strcpy(Str_1_Loc, string_1);
105   Arr_2_Glob[8][7] = 10;
```

```
106
107    __asm__ volatile("li t4, 0x0F0F");
108    for (Run_Index = 1; Run_Index <= NUMBER_OF_RUNS; ++Run_Index)
109    {
110  #ifdef CONTEXT_SWITCH
111      mecall(4U);
112  #endif
113      Proc_5();
114  #ifdef CONTEXT_SWITCH
115      mecall(3U);
116  #endif
117      Proc_4();
118      /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
119      Int_1_Loc = 2;
120      Int_2_Loc = 3;
121      m_strcpy(Str_2_Loc, string_2);
122      //    m_strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
123      Enum_Loc = Ident_2;
124      Bool_Glob = !Func_2(Str_1_Loc, Str_2_Loc);
125      /* Bool_Glob == 1 */
126      while (Int_1_Loc < Int_2_Loc) /* loop body executed once */
127      {
128        Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
129        /* Int_3_Loc == 7 */
130  #ifdef CONTEXT_SWITCH
131        mecall(6U);
132  #endif
133        Proc_7(Int_1_Loc, Int_2_Loc, &Int_3_Loc);
134        /* Int_3_Loc == 7 */
135        Int_1_Loc += 1;
136      } /* while */
137        /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
138  #ifdef CONTEXT_SWITCH
139      mecall(7U);
140  #endif
141      Proc_8(Arr_1_Glob, Arr_2_Glob, Int_1_Loc, Int_3_Loc);
142      /* Int_Glob == 5 */
143  #ifdef CONTEXT_SWITCH
144      mecall(0U);
145  #endif
146      Proc_1(Ptr_Glob);
```

```
147
148      for (Ch_Index = 'A'; Ch_Index <= Ch_2_Glob; ++Ch_Index)
149      /* loop body executed twice */
150      {
151        if (Enum_Loc == Func_1(Ch_Index, 'C'))
152        /* then, not executed */
153        {
154  #ifdef CONTEXT_SWITCH
155          mecall(5U);
156  #endif
157          Proc_6(Ident_1, &Enum_Loc);
158          m_strcpy(Str_2_Loc, string_3);
159          // m_strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 3'RD STRING");
160          Int_2_Loc = Run_Index;
161          Int_Glob = Run_Index;
162        }
163      }
164      /* Int_1_Loc == 3, Int_2_Loc == 3, Int_3_Loc == 7 */
165      Int_2_Loc = Int_2_Loc * Int_1_Loc;
166      Int_1_Loc = Int_2_Loc / Int_3_Loc;
167      Int_2_Loc = 7 * (Int_2_Loc - Int_3_Loc) - Int_1_Loc;
168      /* Int_1_Loc == 1, Int_2_Loc == 13, Int_3_Loc == 7 */
169  #ifdef CONTEXT_SWITCH
170      mecall(1U);
171  #endif
172      Proc_2(&Int_1_Loc);
173      /* Int_1_Loc == 5 */
174    } /* loop "for Run_Index" */
175    __asm__ volatile("li t4, 0xF0F0");
176  }
177
178  void Proc_1(REG Rec_Pointer Ptr_Val_Par)
179  {
180    REG Rec_Pointer Next_Record = Ptr_Val_Par->Ptr_Comp;
181    /* == Ptr_Glob_Next */
182    /* Local variable, initialized with Ptr_Val_Par->Ptr_Comp,    */
183    /* corresponds to "rename" in Ada, "with" in Pascal           */
184
185    structassign(*Ptr_Val_Par->Ptr_Comp, *Ptr_Glob);
186    Ptr_Val_Par->variant.var_1.Int_Comp = 5;
187    Next_Record->variant.var_1.Int_Comp =
```

```
188         Ptr_Val_Par->variant.var_1.Int_Comp;
189     Next_Record->Ptr_Comp = Ptr_Val_Par->Ptr_Comp;
190     Proc_3(&Next_Record->Ptr_Comp);
191     /* Ptr_Val_Par->Ptr_Comp->Ptr_Comp
192                         == Ptr_Glob->Ptr_Comp */
193     if (Next_Record->Discr == Ident_1)
194     /* then, executed */
195     {
196       Next_Record->variant.var_1.Int_Comp = 6;
197       Proc_6(Ptr_Val_Par->variant.var_1.Enum_Comp,
198              &Next_Record->variant.var_1.Enum_Comp);
199       Next_Record->Ptr_Comp = Ptr_Glob->Ptr_Comp;
200       Proc_7(Next_Record->variant.var_1.Int_Comp, 10,
201              &Next_Record->variant.var_1.Int_Comp);
202     }
203     else /* not executed */
204       structassign(*Ptr_Val_Par, *Ptr_Val_Par->Ptr_Comp);
205   } /* Proc_1 */
206
207   void Proc_2(One_Fifty* Int_Par_Ref)
208   /******************/
209   /* executed once */
210   /* *Int_Par_Ref == 1, becomes 4 */
211   {
212     One_Fifty Int_Loc;
213     Enumeration Enum_Loc;
214
215     Int_Loc = *Int_Par_Ref + 10;
216     do /* executed once */
217       if (Ch_1_Glob == 'A')
218       /* then, executed */
219       {
220         Int_Loc -= 1;
221         *Int_Par_Ref = Int_Loc - Int_Glob;
222         Enum_Loc = Ident_1;
223       }                         /* if */
224     while (Enum_Loc != Ident_1); /* true */
225   } /* Proc_2 */
226
227   void Proc_3(Rec_Pointer* Ptr_Ref_Par)
228   /******************/
```

```
229 /* executed once */
230 /* Ptr_Ref_Par becomes Ptr_Glob */
231 {
232   if (Ptr_Glob != Null)
233     /* then, executed */
234     *Ptr_Ref_Par = Ptr_Glob->Ptr_Comp;
235   Proc_7(10, Int_Glob, &Ptr_Glob->variant.var_1.Int_Comp);
236 } /* Proc_3 */
237
238 void Proc_4() /* without parameters */
239 /*******/
240 /* executed once */
241 {
242   Boolean Bool_Loc;
243
244   Bool_Loc = Ch_1_Glob == 'A';
245   Bool_Glob = Bool_Loc | Bool_Glob;
246   Ch_2_Glob = 'B';
247 } /* Proc_4 */
248
249 void Proc_5() /* without parameters */
250 /*******/
251 /* executed once */
252 {
253   Ch_1_Glob = 'A';
254   Bool_Glob = false;
255 } /* Proc_5 */
256
257 // copy word by word
258 char* m_strcpy(char* destination, const char* source)
259 {
260   int fast = 31 / sizeof(uint32_t) + 1;
261   int offset = (fast - 1) * sizeof(uint32_t);
262   int current_block = 0;
263
264   uint32_t* lptr0 = (uint32_t*)destination;
265   uint32_t* lptr1 = (uint32_t*)source;
266
267   while (current_block < fast - 1)
268   {
269     lptr0[current_block] = lptr1[current_block];
```

```
270       ++current_block;
271     }
272     while (offset < 31)
273     {
274       destination[offset] = source[offset];
275       ++offset;
276     }
277     return destination;
278   }
279
280   /*************************************************************************
281    * Function m_malloc() - provides a dynamic memory allocation service
282    * sufficient for two allocations of 50 bytes each or less.
283    */
284   void* m_malloc_1(int nbytes)
285   {
286     static char space_1[100];
287     static char* ptr_1;
288     static char* result;
289     ptr_1 = &space_1[0];
290     result = ptr_1;
291     nbytes = (nbytes & 0xFFFC) + 4;
292     ptr_1 += nbytes;
293
294     return (result);
295   }
296
297   void* m_malloc_2(int nbytes)
298   {
299     static char space_2[100];
300     static char* ptr_2;
301     static char* result_2;
302     ptr_2 = &space_2[0];
303     result_2 = ptr_2;
304     nbytes = (nbytes & 0xFFFC) + 4;
305     ptr_2 += nbytes;
306
307     return (result_2);
308   }
309
310   #ifdef NOSTRUCTASSIGN
```

```
311 void memcpy(register char* d, register char* s, register int l)
312 {
313   int fast = l / sizeof(uint32_t) + 1;
314   int offset = (fast - 1) * sizeof(uint32_t);
315   int current_block = 0;
316   uint32_t* lptr0 = (uint32_t*)d;
317   uint32_t* lptr1 = (uint32_t*)s;
318 
319   while (current_block < fast - 1)
320   {
321     lptr0[current_block] = lptr1[current_block];
322     ++current_block;
323   }
324 
325   while (offset < 31)
326   {
327     d[offset] = s[offset];
328     ++offset;
329   }
330 }
331 #endif
332 
333 void init_string_some(void)
334 {
335   string_some[0] = 'D';
336   string_some[1] = 'H';
337   string_some[2] = 'R';
338   string_some[3] = 'Y';
339   string_some[4] = 'S';
340   string_some[5] = 'T';
341   string_some[6] = 'O';
342   string_some[7] = 'N';
343   string_some[8] = 'E';
344   string_some[9] = ' ';
345   string_some[10] = 'P';
346   string_some[11] = 'R';
347   string_some[12] = 'O';
348   string_some[13] = 'G';
349   string_some[14] = 'R';
350   string_some[15] = 'A';
351   string_some[16] = 'M';
```

```
352   string_some[17] = ',';
353   string_some[18] = ' ';
354   string_some[19] = 'S';
355   string_some[20] = 'O';
356   string_some[21] = 'M';
357   string_some[22] = 'E';
358   string_some[23] = ' ';
359   string_some[24] = 'S';
360   string_some[25] = 'T';
361   string_some[26] = 'R';
362   string_some[27] = 'I';
363   string_some[28] = 'N';
364   string_some[29] = 'G';
365   string_some[30] = '\0';
366 }
367
368 void init_string_1(void)
369 {
370   string_1[0] = 'D';
371   string_1[1] = 'H';
372   string_1[2] = 'R';
373   string_1[3] = 'Y';
374   string_1[4] = 'S';
375   string_1[5] = 'T';
376   string_1[6] = 'O';
377   string_1[7] = 'N';
378   string_1[8] = 'E';
379   string_1[9] = ' ';
380   string_1[10] = 'P';
381   string_1[11] = 'R';
382   string_1[12] = 'O';
383   string_1[13] = 'G';
384   string_1[14] = 'R';
385   string_1[15] = 'A';
386   string_1[16] = 'M';
387   string_1[17] = ',';
388   string_1[18] = ' ';
389   string_1[19] = '1';
390   string_1[20] = '\'';
391   string_1[21] = 'S';
392   string_1[22] = 'T';
```

```
393     string_1[23] = ' ';
394     string_1[24] = 'S';
395     string_1[25] = 'T';
396     string_1[26] = 'R';
397     string_1[27] = 'I';
398     string_1[28] = 'N';
399     string_1[29] = 'G';
400     string_1[30] = '\0';
401   }
402   void init_string_2(void)
403   {
404     string_2[0] = 'D';
405     string_2[1] = 'H';
406     string_2[2] = 'R';
407     string_2[3] = 'Y';
408     string_2[4] = 'S';
409     string_2[5] = 'T';
410     string_2[6] = 'O';
411     string_2[7] = 'N';
412     string_2[8] = 'E';
413     string_2[9] = ' ';
414     string_2[10] = 'P';
415     string_2[11] = 'R';
416     string_2[12] = 'O';
417     string_2[13] = 'G';
418     string_2[14] = 'R';
419     string_2[15] = 'A';
420     string_2[16] = 'M';
421     string_2[17] = ',';
422     string_2[18] = ' ';
423     string_2[19] = '2';
424     string_2[20] = '\'';
425     string_2[21] = 'N';
426     string_2[22] = 'D';
427     string_2[23] = ' ';
428     string_2[24] = 'S';
429     string_2[25] = 'T';
430     string_2[26] = 'R';
431     string_2[27] = 'I';
432     string_2[28] = 'N';
433     string_2[29] = 'G';
```

```
434   string_2[30] = '\0';
435 }
436 void init_string_3(void)
437 {
438   string_3[0] = 'D';
439   string_3[1] = 'H';
440   string_3[2] = 'R';
441   string_3[3] = 'Y';
442   string_3[4] = 'S';
443   string_3[5] = 'T';
444   string_3[6] = 'O';
445   string_3[7] = 'N';
446   string_3[8] = 'E';
447   string_3[9] = ' ';
448   string_3[10] = 'P';
449   string_3[11] = 'R';
450   string_3[12] = 'O';
451   string_3[13] = 'G';
452   string_3[14] = 'R';
453   string_3[15] = 'A';
454   string_3[16] = 'M';
455   string_3[17] = ',';
456   string_3[18] = ' ';
457   string_3[19] = '3';
458   string_3[20] = '\'';
459   string_3[21] = 'R';
460   string_3[22] = 'D';
461   string_3[23] = ' ';
462   string_3[24] = 'S';
463   string_3[25] = 'T';
464   string_3[26] = 'R';
465   string_3[27] = 'I';
466   string_3[28] = 'N';
467   string_3[29] = 'G';
468   string_3[30] = '\0';
469 }
```

B.1.: dhry_1.c

```
1 /*
```

```
2   ****************************************************************************
3   *
4   *                    "DHRYSTONE" Benchmark Program
5   *                    -----------------------------
6   *
7   *  Version:    C, Version 2.1
8   *
9   *  File:       dhry_2.c (part 3 of 3)
10  *
11  *  Date:       May 25, 1988
12  *
13  *  Author:     Reinhold P. Weicker
14  *
15  ****************************************************************************
16  */
17
18 #include "dhry.h"
19 #include "../common.h"
20
21 #ifndef REG
22 #define REG
23 /* REG becomes defined as empty */
24 /* i.e. no register variables   */
25 #endif
26
27 Boolean Func_3(Enumeration);
28 int m_strcmp(const char* X, const char* Y);
29
30 extern int Int_Glob;
31 extern char Ch_1_Glob;
32
33 void Proc_6(Enumeration Enum_Val_Par, Enumeration* Enum_Ref_Par)
34 /*********************************/
35 /* executed once */
36 /* Enum_Val_Par == Ident_3, Enum_Ref_Par becomes Ident_2 */
37 {
38   *Enum_Ref_Par = Enum_Val_Par;
39   if (!Func_3(Enum_Val_Par))
40     /* then, not executed */
41     *Enum_Ref_Par = Ident_4;
42   switch (Enum_Val_Par)
```

```
43    {
44    case Ident_1:
45      *Enum_Ref_Par = Ident_1;
46      break;
47    case Ident_2:
48      if (Int_Glob > 100)
49        /* then */
50        *Enum_Ref_Par = Ident_1;
51      else
52        *Enum_Ref_Par = Ident_4;
53      break;
54    case Ident_3: /* executed */
55      *Enum_Ref_Par = Ident_2;
56      break;
57    case Ident_4:
58      break;
59    case Ident_5:
60      *Enum_Ref_Par = Ident_3;
61      break;
62    } /* switch */
63  } /* Proc_6 */
64
65  void Proc_7(One_Fifty Int_1_Par_Val, One_Fifty Int_2_Par_Val,
66              One_Fifty* Int_Par_Ref)
67  /**********************************************/
68  /* executed three times                                      */
69  /* first call:      Int_1_Par_Val == 2, Int_2_Par_Val == 3,  */
70  /*                  Int_Par_Ref becomes 7                    */
71  /* second call:     Int_1_Par_Val == 10, Int_2_Par_Val == 5, */
72  /*                  Int_Par_Ref becomes 17                   */
73  /* third call:      Int_1_Par_Val == 6, Int_2_Par_Val == 10, */
74  /*                  Int_Par_Ref becomes 18                   */
75  {
76    One_Fifty Int_Loc;
77
78    Int_Loc = Int_1_Par_Val + 2;
79    *Int_Par_Ref = Int_2_Par_Val + Int_Loc;
80  } /* Proc_7 */
81
82  void Proc_8(Arr_1_Dim Arr_1_Par_Ref, Arr_2_Dim Arr_2_Par_Ref,
83              int Int_1_Par_Val, int Int_2_Par_Val)
```

```
84   /*********************************************************************/
85   /* executed once      */
86   /* Int_Par_Val_1 == 3 */
87   /* Int_Par_Val_2 == 7 */
88   {
89     REG One_Fifty Int_Index;
90     REG One_Fifty Int_Loc;
91
92     Int_Loc = Int_1_Par_Val + 5;
93     Arr_1_Par_Ref[Int_Loc] = Int_2_Par_Val;
94     Arr_1_Par_Ref[Int_Loc + 1] = Arr_1_Par_Ref[Int_Loc];
95     Arr_1_Par_Ref[Int_Loc + 10] = Int_Loc;
96     for (Int_Index = Int_Loc; Int_Index <= Int_Loc + 1; ++Int_Index)
97       Arr_2_Par_Ref[Int_Loc][Int_Index] = Int_Loc;
98     Arr_2_Par_Ref[Int_Loc][Int_Loc - 1] += 1;
99     Arr_2_Par_Ref[Int_Loc + 10][Int_Loc] = Arr_1_Par_Ref[Int_Loc];
100    Int_Glob = 5;
101  } /* Proc_8 */
102
103  // mark
104  Enumeration Func_1(Capital_Letter Ch_1_Par_Val,
105                     Capital_Letter Ch_2_Par_Val)
106  /**************************************************/
107  /* executed three times                                          */
108  /* first call:      Ch_1_Par_Val == 'H', Ch_2_Par_Val == 'R'    */
109  /* second call:     Ch_1_Par_Val == 'A', Ch_2_Par_Val == 'C'    */
110  /* third call:      Ch_1_Par_Val == 'B', Ch_2_Par_Val == 'C'    */
111
112  {
113    Capital_Letter Ch_1_Loc;
114    Capital_Letter Ch_2_Loc;
115
116    Ch_1_Loc = Ch_1_Par_Val;
117    Ch_2_Loc = Ch_1_Loc;
118    if (Ch_2_Loc != Ch_2_Par_Val)
119      /* then, executed */
120      return (Ident_1);
121    else /* not executed */
122    {
123      Ch_1_Glob = Ch_1_Loc;
124      return (Ident_2);
```

```
125     }
126   } /* Func_1 */
127
128   Boolean Func_2(Str_30 Str_1_Par_Ref, Str_30 Str_2_Par_Ref)
129   /*************************************************/
130   /* executed once */
131   /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
132   /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */
133
134   {
135     REG One_Thirty Int_Loc;
136     Capital_Letter Ch_Loc;
137
138     Int_Loc = 2;
139     while (Int_Loc <= 2) /* loop body executed once */
140       if (Func_1(Str_1_Par_Ref[Int_Loc], Str_2_Par_Ref[Int_Loc + 1]) ==
141           Ident_1)
142       /* then, executed */
143       {
144         Ch_Loc = 'A';
145         Int_Loc += 1;
146       } /* if, while */
147     if (Ch_Loc >= 'W' && Ch_Loc < 'Z')
148       /* then, not executed */
149       Int_Loc = 7;
150     if (Ch_Loc == 'R')
151       /* then, not executed */
152       return (true);
153     else /* executed */
154     {
155       if (m_strcmp(Str_1_Par_Ref, Str_2_Par_Ref) > 0)
156       /* then, not executed */
157       //    if(0)
158       {
159         Int_Loc += 7;
160         Int_Glob = Int_Loc;
161         return (true);
162       }
163       else /* executed */
164         return (false);
165     } /* if Ch_Loc */
```

```
166 } /* Func_2 */
167
168 Boolean Func_3(Enumeration Enum_Par_Val)
169 /***************************/
170 /* executed once         */
171 /* Enum_Par_Val == Ident_3 */
172 {
173   Enumeration Enum_Loc;
174
175   Enum_Loc = Enum_Par_Val;
176   if (Enum_Loc == Ident_3)
177     /* then, executed */
178     return (true);
179   else /* not executed */
180     return (false);
181 } /* Func_3 */
182
183 int m_strcmp(const char* X, const char* Y)
184 {
185   int fast = 31 / sizeof(uint32_t) + 1;
186   int offset = (fast - 1) * sizeof(uint32_t);
187   int current_block = 0;
188
189   if (31 <= sizeof(uint32_t))
190   {
191     fast = 0;
192   }
193
194   uint32_t* lptr0 = (uint32_t*)X;
195   uint32_t* lptr1 = (uint32_t*)Y;
196
197   while (current_block < fast)
198   {
199     if ((lptr0[current_block] ^ lptr1[current_block]))
200     {
201       return (int)(lptr0[current_block] - lptr1[current_block]);
202     }
203     ++current_block;
204   }
205
206   while (31 > offset)
```

```
207     {
208       if ((X[offset] ^ Y[offset]))
209       {
210         return (int)((unsigned char)X[offset] - (unsigned char)Y[offset]);
211       }
212 
213       ++offset;
214     }
215     return 0;
216   }
```

B.2.: dhry_2.c

B.2. Kernel and Exception Handler

```
1   #include "kernel.h"
2   #include "mpu.h"
3 
4   extern void kernel_entry(void);
5 
6   #ifdef MPU_ACTIVE
7   uint32_t mpu_enabled;
8   #endif
9 
10  void exception_handler(uint32_t cause, uint32_t pc, uint32_t value)
11  {
12    switch (cause)
13    {
14    case 1U: // iaf
15      break;
16    case 5U: // laf
17      break;
18    case 7U: // saf
19      break;
20    case 11U: // mecall
21  #ifdef MPU_ACTIVE
22      MPU_load_cfg(value);
23  #endif
```

```
24        break;
25      default:
26        break;
27      }
28      return;
29    }
30
31    void mecall(const uint32_t number)
32    {
33      __asm__ volatile("mv a7, %0" : : "r"(number) : "t4");
34      __asm__ volatile("ecall");
35      return;
36    }
37
38    #define MTVEC_ADDR 0x305
39    #define MSEE_ADDR 0x307
40    #define CTRL_ADDR 0x7C0
41
42    void setup_exceptions(void)
43    {
44      CSR_WRITE((uint16_t*)MTVEC_ADDR, ((uint32_t)&kernel_entry) << 2U);
45      // enable mecall, saf, laf, iaf exceptions
46      CSR_WRITE((uint16_t*)MSEE_ADDR, 0x8A2);
47      return;
48    }
```

B.3.: kernel.c for Dhrystone

```
1     #define cause t0
2     #define handler t2
3     #define mpu_enabled_reg t3
4
5     .extern exception_handler
6
7     #ifdef MPU_ACTIVE
8     .equ MPU_CTRL, 0x7C0
9     .extern mpu_enabled
10    #endif
11    .section .kernel
12    .global kernel_entry
```

```
13 kernel_entry:
14   /* save current context */
15   addi sp,sp,-68
16   sw sp,64(sp)
17   sw ra,60(sp)
18   sw t0,56(sp)
19   sw t1,52(sp)
20   sw t2,48(sp)
21   sw a0,44(sp)
22   sw a1,40(sp)
23   sw a2,36(sp)
24   sw a3,32(sp)
25   sw a4,28(sp)
26   sw a5,24(sp)
27   sw a6,20(sp)
28   sw a7,16(sp)
29   sw t3,12(sp)
30   sw t4,8(sp)
31   sw t5,4(sp)
32   sw t6,0(sp)
33   #ifdef MPU_ACTIVE
34   lui mpu_enabled_reg, %hi(mpu_enabled)
35   addi mpu_enabled_reg, mpu_enabled_reg, %lo(mpu_enabled)
36   csrrci t1, MPU_CTRL, 1
37   sw t1,  0(mpu_enabled_reg)
38   #endif
39   j get_cause
40 .section .text
41 get_cause:
42   csrrw cause, mcause, zero
43   li t1, 0x7FFFFFFF
44   bleu cause, t1, exc_handler
45   j dispatch
46 exc_handler:
47   la handler, exception_handler
48   /* prepare for function call */
49   addi sp, sp, -4
50   sw ra, 0(sp)
51   mv a0, cause
52   csrrw a1, mepc, zero /* pass offending PC as function parameter */
53   mv a2, a7
```

```
54    jalr handler /* call the handler */
55    lw ra, 0(sp)
56    addi sp, sp, 4
57    j dispatch
58  .section .kernel
59  dispatch:
60    #ifdef MPU_ACTIVE
61    lui mpu_enabled_reg, %hi(mpu_enabled)
62    addi mpu_enabled_reg, mpu_enabled_reg, %lo(mpu_enabled)
63    lw t1,  0(mpu_enabled_reg)
64    csrw MPU_CTRL, t1
65    #endif
66    /* restore context */
67    lw sp,64(sp)
68    lw ra,60(sp)
69    csrw mepc, ra /* setup address to return to */
70    lw t0,56(sp)
71    lw t1,52(sp)
72    lw t2,48(sp)
73    lw a0,44(sp)
74    lw a1,40(sp)
75    lw a2,36(sp)
76    lw a3,32(sp)
77    lw a4,28(sp)
78    lw a5,24(sp)
79    lw a6,20(sp)
80    lw a7,16(sp)
81    lw t3,12(sp)
82    lw t4,8(sp)
83    lw t5,4(sp)
84    lw t6,0(sp)
85    addi sp,sp,68
86    mret
```

B.4.: kernel.S for Dhrystone

B.3. MPU Management

```
#ifndef _MPU_H_
#define _MPU_H_

#include "common.h"

typedef uint32_t MPU_addr_t;

typedef union MPU_cfg
{
  uint32_t cfg;
  struct
  {
    uint8_t cfg0;
    uint8_t cfg1;
    uint8_t cfg2;
    uint8_t cfg3;
  };
} MPU_cfg_t;

typedef struct MPU_region
{
  MPU_cfg_t cfg;
  uint32_t num_of_addresses;
  MPU_addr_t addr[4];
} MPU_region_t;

void setup_MPU_regions(const uint32_t num_of_regions,
                       const uint32_t num_of_cfgs);
void MPU_load_cfg(const uint32_t ID);

#endif /* _MPU_H_ */
```

B.5.: mpu.h

```
#include "mpu.h"

#define MPU_STATUS 0xFC0
```

```
4  #define MPU_CTRL 0x7C0
5  #define MPU_CFG0 0x7C1
6  #define MPU_CFG1 0x7C2
7  #define MPU_CFG2 0x7C3
8  #define MPU_CFG3 0x7C4
9  #define MPU_CFG4 0x7C5
10 #define MPU_CFG5 0x7C6
11 #define MPU_CFG6 0x7C7
12 #define MPU_CFG7 0x7C8
13 #define MPU_ADDR0 0x7C9
14 #define MPU_ADDR1 0x7CA
15 #define MPU_ADDR2 0x7CB
16 #define MPU_ADDR3 0x7CC
17 #define MPU_ADDR4 0x7CD
18 #define MPU_ADDR5 0x7CE
19 #define MPU_ADDR6 0x7CF
20 #define MPU_ADDR7 0x7D0
21 #define MPU_ADDR8 0x7D1
22 #define MPU_ADDR9 0x7D2
23 #define MPU_ADDR10 0x7D3
24 #define MPU_ADDR11 0x7D4
25 #define MPU_ADDR12 0x7D5
26 #define MPU_ADDR13 0x7D6
27 #define MPU_ADDR14 0x7D7
28 #define MPU_ADDR15 0x7D8
29 #define MPU_ADDR16 0x7D9
30 #define MPU_ADDR17 0x7DA
31 #define MPU_ADDR18 0x7DB
32 #define MPU_ADDR19 0x7DC
33 #define MPU_ADDR20 0x7DD
34 #define MPU_ADDR21 0x7DE
35 #define MPU_ADDR22 0x7DF
36 #define MPU_ADDR23 0x7E0
37 #define MPU_ADDR24 0x7E1
38 #define MPU_ADDR25 0x7E2
39 #define MPU_ADDR26 0x7E3
40 #define MPU_ADDR27 0x7E4
41 #define MPU_ADDR28 0x7E5
42 #define MPU_ADDR29 0x7E6
43 #define MPU_ADDR30 0x7E7
44 #define MPU_ADDR31 0x7E8
```

```
45
46  #define CSR_WRITE(addr, value)                                    \
47    __asm__ volatile("csrw %0, %1" : : "i"(addr), "r"(value))
48
49  static MPU_region_t MPU_cfgs[8][8]
50      __attribute__((section(".MPU_CFGS")));
51  static uint32_t MPU_cfgs_size;
52
53  void MPU_write_cfg_reg(const uint32_t number, const uint32_t value);
54  void MPU_write_addr_reg(const uint32_t number, const uint32_t value);
55
56  void MPU_write_addr_reg(const uint32_t number, const uint32_t value)
57  {
58    switch (number)
59    {
60    case 0:
61      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR0), "r"(value));
62      break;
63    case 1:
64      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR1), "r"(value));
65      break;
66    case 2:
67      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR2), "r"(value));
68      break;
69    case 3:
70      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR3), "r"(value));
71      break;
72    case 4:
73      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR4), "r"(value));
74      break;
75    case 5:
76      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR5), "r"(value));
77      break;
78    case 6:
79      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR6), "r"(value));
80      break;
81    case 7:
82      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR7), "r"(value));
83      break;
84    case 8:
85      __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR8), "r"(value));
```

```
86          break;
87        case 9:
88          __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR9), "r"(value));
89          break;
90        case 10:
91          __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR10), "r"(value));
92          break;
93        case 11:
94          __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR11), "r"(value));
95          break;
96        case 12:
97          __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR12), "r"(value));
98          break;
99        case 13:
100         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR13), "r"(value));
101         break;
102       case 14:
103         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR14), "r"(value));
104         break;
105       case 15:
106         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR15), "r"(value));
107         break;
108       case 16:
109         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR16), "r"(value));
110         break;
111       case 17:
112         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR17), "r"(value));
113         break;
114       case 18:
115         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR18), "r"(value));
116         break;
117       case 19:
118         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR19), "r"(value));
119         break;
120       case 20:
121         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR20), "r"(value));
122         break;
123       case 21:
124         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR21), "r"(value));
125         break;
126       case 22:
```

```
127         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR22), "r"(value));
128         break;
129     case 23:
130         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR23), "r"(value));
131         break;
132     case 24:
133         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR24), "r"(value));
134         break;
135     case 25:
136         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR25), "r"(value));
137         break;
138     case 26:
139         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR26), "r"(value));
140         break;
141     case 27:
142         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR27), "r"(value));
143         break;
144     case 28:
145         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR28), "r"(value));
146         break;
147     case 29:
148         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR29), "r"(value));
149         break;
150     case 30:
151         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR30), "r"(value));
152         break;
153     case 31:
154         __asm__ volatile("csrw %0, %1" : : "i"(MPU_ADDR31), "r"(value));
155         break;
156     default:
157         break;
158     }
159 }
160 
161 void MPU_write_cfg_reg(const uint32_t number, const uint32_t value)
162 {
163     switch (number)
164     {
165     case 0:
166         __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG0), "r"(value));
167         break;
```

```
168    case 1:
169      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG1), "r"(value));
170      break;
171    case 2:
172      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG2), "r"(value));
173      break;
174    case 3:
175      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG3), "r"(value));
176      break;
177    case 4:
178      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG4), "r"(value));
179      break;
180    case 5:
181      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG5), "r"(value));
182      break;
183    case 6:
184      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG6), "r"(value));
185      break;
186    case 7:
187      __asm__ volatile("csrw %0, %1" : : "i"(MPU_CFG7), "r"(value));
188      break;
189    default:
190      break;
191    }
192  }
193
194  void MPU_load_cfg(const uint32_t ID)
195  {
196    if (MPU_cfgs && MPU_cfgs[ID])
197    {
198      MPU_region_t* new_cfg = MPU_cfgs[ID];
199      uint32_t address_counter = 0;
200      for (uint32_t i = 0; i < MPU_cfgs_size; i++)
201      {
202        /* write MPU configuration registers
203        MPU_write_cfg_reg(uint32_t number, uint32_t value) */
204        MPU_write_cfg_reg(i, new_cfg[i].cfg.cfg);
205        for (uint32_t j = 0; j < new_cfg[i].num_of_addresses; j++)
206        {
207          /* write MPU address registers
208          MPU_write_addr_reg(uint32_t number, uint32_t value) */
```

```
209          MPU_write_addr_reg(address_counter++, new_cfg[i].addr[j]);
210        }
211      }
212    }
213    return;
214  }
215
216  /* fill MPU_cfgs with dummy values */
217  void setup_MPU_regions(const uint32_t num_of_regions,
218                         const uint32_t num_of_cfgs)
219  {
220    if (num_of_regions > 0 && num_of_cfgs > 0)
221    {
222      MPU_cfgs_size = 0U;
223      uint32_t non_multiple_of_4 = num_of_regions % 4U;
224      uint32_t fully_filled_regions = num_of_regions / 4U;
225      for (uint32_t cfg_counter = 0U; cfg_counter < num_of_cfgs;
226           cfg_counter++)
227      {
228        if (non_multiple_of_4 > 0)
229        {
230          switch (non_multiple_of_4)
231          {
232          case 1U:
233            MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg0 =
234                cfg_counter;
235            MPU_cfgs[cfg_counter][num_of_regions - 1U].num_of_addresses =
236                non_multiple_of_4;
237            MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[0U] =
238                cfg_counter;
239            break;
240          case 2U:
241            MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg0 =
242                cfg_counter;
243            MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg1 =
244                cfg_counter;
245            MPU_cfgs[cfg_counter][num_of_regions - 1U].num_of_addresses =
246                non_multiple_of_4;
247            MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[0U] =
248                cfg_counter;
249            MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[1U] =
```

```
250                 cfg_counter;
251             break;
252           case 3U:
253             MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg0 =
254                 cfg_counter;
255             MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg1 =
256                 cfg_counter;
257             MPU_cfgs[cfg_counter][num_of_regions - 1U].cfg.cfg2 =
258                 cfg_counter;
259             MPU_cfgs[cfg_counter][num_of_regions - 1U].num_of_addresses =
260                 non_multiple_of_4;
261             MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[0U] =
262                 cfg_counter;
263             MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[1U] =
264                 cfg_counter;
265             MPU_cfgs[cfg_counter][num_of_regions - 1U].addr[2U] =
266                 cfg_counter;
267             break;
268           default:
269             break;
270           };
271           MPU_cfgs_size = 1U;
272         }
273         for (uint32_t i = 0; i < fully_filled_regions; i++)
274         {
275           MPU_cfgs[cfg_counter][i].cfg.cfg = cfg_counter;
276           MPU_cfgs[cfg_counter][i].num_of_addresses = 4U;
277           MPU_cfgs[cfg_counter][i].addr[0U] = 0xDEADBEEF + cfg_counter;
278           MPU_cfgs[cfg_counter][i].addr[1U] = 0xBEEFDEAD + cfg_counter;
279           MPU_cfgs[cfg_counter][i].addr[2U] = 0xBAADBEEF + cfg_counter;
280           MPU_cfgs[cfg_counter][i].addr[3U] = 0xBEEFBAAD + cfg_counter;
281         }
282       }
283       MPU_cfgs_size += fully_filled_regions;
284     }
285   }
```

B.6.: mpu.c