POLITECNICO DI TORINO

Collegio di Ingegneria Informatica,
del Cinema e Meccatronica

Master degree course in Computer Engineering

Master Degree Thesis

# Design and programming of a coprocessor for a RISC-V architecture

Guidelines for embedding computing cores as RISC-V coprocessors


**Supervisor:**
prof. Massimo Poncino

**Candidate**
Davide Pala

**Company supervisor**
**CEA - Grenoble (FR)**
dott. ing. Ivan Miro Panades

Academic year 2016-2017

# Summary

The extraordinary development of the embedded systems market and the ever increasing interest in the IoT domain, is leading the electronic design towards the development of ultra-low power and ultra-low energy systems. Aware of this scenario the CEA LETI research center is working on a novel IoT platform with the objective of achieving efficient computations without sacrificing the performance. This project called L-IoT platform, realises a flexible fully integrated system, with the use of a partitioned architecture. The design is divided into an always-on, ultra-low power and ultra-low energy system for the management of low intensity activities, and an on-demand part which handles the more computationally intensive applications. In this context the advent of SoCs and dedicated hardware is playing an important role in achieving a balance between the power consumption and the performance trade-off. From these premises rises the interest for the study of coprocessors for the RISC-V core which powers the on-demand section of the L-IoT architecture. This work is thus inserted in the wider picture of the L-IoT platform, in particular it focuses on the design and development of coprocessors for the on-demand part of the system. This thesis presents an investigation on the coprocessor development, starting from an analysis of the interfaces, with a latency study. The latency analysis results shows an high latency overhead for the transfers of data between the coprocessor and the core, wich excluded the use of this interface for tight coupled coprocessors. Instead the decoupled protocol of the interfaces and the presence of a direct channel to the L1 data cache, suggested the use of this interface for decoupled throughput coprocessors. In particular the analysis of the memory interface shows the availability of a 50% throughput on the data cache channel. Considering the results of the analysis, a design for a cryptographic coprocessor is proposed. This design makes use of a block-cipher IP internaly developed in the CEA LETI laboratory, implementing both AES and PRESENT encryption. This module is used inside the coprocessor, the proposed design implements an architecture capable of performing both AES and PRESENT encryptions in both ECB and OFB mode of operation. In particular the OFB mode allows a completely autonomous execution which exploits

the memory link of the coprocessor interface. With this mode the coprocessor is able to achieve the maximum throughput allowed by the block-cipher core, with just a small programming overhead to set up the operation. The accelerator is programmed in a peripheral-like fashion, with just the use of read and write operations on the configuration registers. Combining the case study of the cryptographic accelerator and the results of the latency study on the interface, a generic framework for the development of coprocessors is outlined. In particular this documents tries to reports the common recurring patterns that can be identified in the development of different types of accelerators. It is presented the possibility of a universal extensible interface module, implementing the same read/write programming model, independent on the type of coprocessor and with a common interface on the register file. Various considerations on different design aspects are presented, as the general architectural framework, the register bank organisation and the possible extensions of the coprocessor FSM. The programming aspect is also considered, with an overview of the possible ISA models and the different addressing modes. In the end some considerations on performances are presented. The identification of the problems caused by the coprocessor interface latency, lead to the suggestion of avoiding the tight coupled coprocessors, in favour of the ones based on throughput applications, or performing long latency operations and not requiring frequent communications with the CPU.

# Acknowledgements

This thesis is the result of a complex work, carried out during a six month internship in the CEA-LETI research center of Grenoble. This internship was a great experience, it allowed me to live in the lovely city of Grenoble and to work in an international research center, alongside with researchers and engineers from all around the world. It was a great opportunity to see and work with the tools and practices of the silicon industry, and to experience the methods of research and how its world works.

I would like to thank CEA-LETI, and the LISAN laboratory in particular, for this opportunity that I was offered.

I would like to tank my internship tutor Ivan for all the support and the suggestions during the whole stage period and throughout the writing of this thesis. This work would not be possible without his guidance and his tips and advice, with which I learned some of the tricks of the trade.

I also thank all the people in the lab which were always ready to help and share their experience. In particular I'd like to tank Hieu and Thanos for the help with the block-cipher core, David for his support when I used his register file generator, Eric, Romain and Cesar for their help in configuring the software environment.

A special thanks to all the young guys in the lab, I'm hoping and looking forward for a new lan-party!

# Ringraziamenti

Questa tesi è il frutto di un lungo lavoro maturato in una lunga eperienza in Francia, per la quale devo ringraziare tante persone che, in modi diversi, mi hanno aiutato e sostenuto.

Desidero ringraziare innanzi tutto il professore Massimo Poncino, Relatore di questa tesi, senza i cui preziosi consigli non avrei intrapreso questa esperienza. Ringrazio il professore anche per la disponibilità e per avermi aiutato, con i suoi suggerimenti, nelle scelte sul mio futuro lavorativo.

Ringrazio Andrea, che mi ha aiutato dapprima a districarmi nella giungla burocratica e, in seguito, ad ambientarmi a Grenoble e nel laboratorio.

Ringrazio Smeralda che, oltre a sopportare il mio caratteraccio, ha posto un argine alle mie stravaganti scelte cromatiche nelle figure di questa tesi.

Ringrazio Ronald per avermi aiutato a impostare questo documento, dandomi il suo esempio e il suo supporto.

Ringrazio la mia famiglia, soprattutto Fabio, Mamma e Babbo per avermi sostenuto e consigliato, sopportando, più di tutti, il mio carattere particolare e il mio essere un figlio, un nipote e un fratello difficile e spesso un po' distante.

Ringrazio il Collegio Einaudi per essere stato la mia Casa in questi anni di studi e saluto tutti i "coinquilini" passati e presenti, con cui ho condiviso questa Casa.

Ringrazio e saluto tutti gli amici conterranei a Torino, compagni nel “distérru”, che hanno animato questi anni da fuori sede.

Ringrazio e saluto tutti gli amici di Ozieri che sono sempre un ottimo motivo per tornare in “bidda”.

# Contents

# List of Tables

# List of Figures

# Introduction

## Coprocessor

In the IT field, the term coprocessor, also called accelerator, is often used to indicate any kind of special purpose circuit coupled to a processing unit. They are generally used to help a general purpose processor to deal with a restricted set of tasks, in which the coprocessor is specialised.

Dedicated processors are generally used to improve the performance of a computing system, by accelerating tasks that are highly computing intensive or too slow to be performed by software routines running on a general purpose processor. Implementing in hardware these functions can lead to many benefits beyond the sheer improvement of performance. For example the reduced power consumption, that usually comes from the use of dedicated circuits, and the possibility of leaving the main processor free to execute other tasks.

In general the term coprocessor has been used during the years to refer to a wide variety of digital circuits, ranging from dedicated arithmetic units embedded in the pipeline of the processor, as in the case of some Floating Point Units (FPUs), to peripheral cores or external devices such as graphic processors. In the past some of this devices, like arithmetic cores or even FPUs, used to be external chips used to expand the capabilities of a system. Nowadays many of these functions are integrated in the instruction sets and are often embedded in the pipeline of the processors, moreover in modern System on Chips (SoCs) some commonly used computing cores are often integrated in the same die of the processor.

This work takes in consideration a specific Central Processing Unit (CPU) called *Rocket core*. This CPU implements the *RISC-V* Instruction Set Architecture (ISA)[1] and enables the presence of coprocessors through the use of a dedicated interface named **RoCC**.

In the context of this document the therms "coprocessor" and "accelerator" are used referring to any circuit attached to the Rocket core through the use of the RoCC coprocessor interface.

# The context

This thesis is the result of a work conducted in the CEA LETI laboratory of Grenoble, in the context of a wider project called Low-power Internet of Things (L-IoT) Platform. The L-IoT project aims at obtaining a system platform able to cover a wide variety of applications in the field of Internet of Things (IoT). Therefore this system is designed to target a wide variety of energy/performance requirements, ranging from ultra low energy and low performance applications, such as tracking and monitoring, to applications like video surveillance that requires higher performance and have higher power consumption.

This platform has been thought as being fully integrated, and is composed of two main sub-systems: an always-on part which is meant to handle the normal operations and a on-demand sub-system that is waken up only for handling the tasks that requires higher performance. The always-responsive sub-system embeds advanced wake-up features and is able to work in the $pW$ to $\mu W$ range of power consumption.

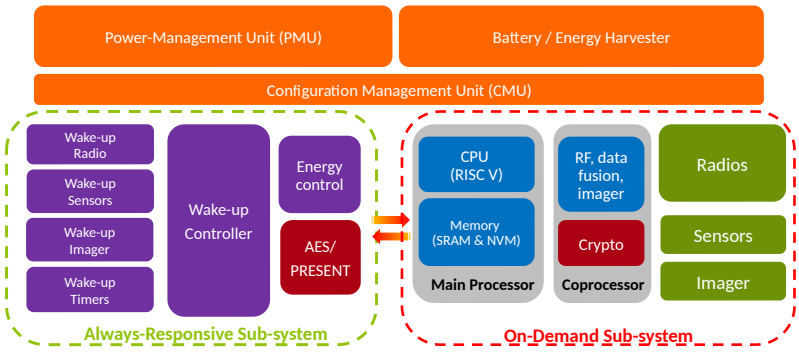


Figure 1. Scheme of the L-IoT architecture

On the other hand the on-demand sub-system is thought for the tasks requiring higher performance and is meant to work with a power consumption in the $mW$ range. In order to answer these strict needs for low power and relatively high performance the on-demand part will integrate a RISC-V based processor assisted by some dedicated coprocessors. This architecture takes advantage of the parallelism between the core that performs a task and the accelerator that executes another to reduce the compute time and thus the energy.

One of the CPUs taken into account for this sub-system is the *Rocket core* processor described in Chapter 1.

# The work-flow

The work described in this document was carried out within a development framework which resulted from the combination of several tools. In particular this thesis focuses on the hardware design aspects of the development of coprocessors. However in order to develop this work, a whole hardware/software infrastructure has been organised. The Figure 2 shows the main components that concurred in the creation of the framework for this work. In particular the vertical sections regards the hardware design and generation, while the horizontal one shows the software compilation chain. This work mainly focuses on the green section,

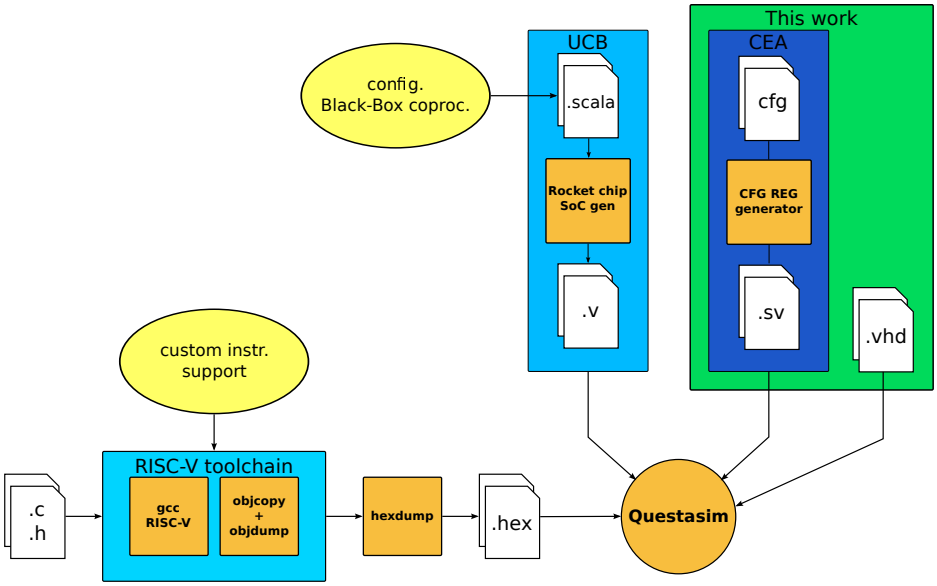


Figure 2. Work-flow diagram for the development of this thesis.

which represents the hardware coprocessor design process. Overall the combination of the UCB framework for the generation of the core, the GNU tool-chain for the compilation of the software and the hardware design process for the coprocessor composes the picture in which this work was devised. The appendices Appendix B, Appendix C and Appendix D reports the steps required to set up and use this work-flow.

## Thesis objective

In this picture (the L-IoT platform and the work-flow) this thesis focuses on the design and programming of coprocessors for the RoCC interface, targeting modularity and the use of stand-alone cores not specifically designed for being part of a coprocessor. This work answer the need of studying and understanding the Rocket core and the RoCC interface and exploring the possibility of enhancing the core with the use of accelerators. It also tries to cover the lack of documentation on the subject and presents a new approach for the integration of coprocessors with this interface.

The objective is to give guidelines for the development of accelerators and define a design framework generic with respect to the coprocessor function. An analysis of the performance of the RoCC interface is presented, taking into account the overhead and the latency introduced by the communication between the core and the coprocessor. As a case study the implementation of a cryptographic accelerator is presented, using the block-cipher module described in [3] and [4]. This block-cipher module is used as starting point for an extended architecture that implements the OFB mode of encryption. This architectural choice enables the accelerator to perform both encryption and decryption, moreover it allows the coprocessor to cipher a plain-text of generic length. When running in this mode of operation the core is left free to run other tasks and the only overhead is the small time spent for programming the coprocessor before the encryption.

Starting from the analysis of the interface and from the blockcipher design example a generic design approach is outlined, identifying the common recurrent elements applicable to different kind of coprocessors, and the one that are specific to each design.

This thesis is organised in the following chapters:

- **Chapter 1** contains the basic background information regarding the RISC-V ISA, the Rocket chip generator and the Rocket core which are needed for understanding the platform on top of which this work is developed.
- **Chapter 2** explains the RoCC interface and its main signals, providing an analysis of the latency and some possible improvement.
- **Chapter 3** presents an implementation case study of a block-cipher coprocessor, starting from an already existing block-cipher core, enhancing it to support new features and embedding it in a RoCC coprocessor design.
- **Chapter 4** outlines the considerations and the architectural choices for a generic coprocessor development.
- **Chapter 5** draws the conclusions on the work.

In addition the appendices serves as documentation for setting up the working environment, whit the aim of easing the integration of the Rocket Core in a industrial development flow, taking in consideration the tools used in the CEA LETI laboratory.

# Chapter 1

# RISC-V and Rocket-core overview

This chapter introduces the background information useful for understanding the context of and the platforms on top of which this work is developed. The first section presents an overview of the RISC-V ISA, introducing its main design goals and principles, describing the instruction formats and the extension mechanism. Then in the second section the *Rocket Core* processor together with the *Rocket Chip* SoC generator are introduced, presenting some of their main features and describing the design flow.

## 1.1 RISC-V an open ISA

RISC-V is a new and open ISA, that was initially developed by the University of California Berkeley (UCB) and it is now managed by the RISC-V Foundation[1]. Designed following the Reduced Instruction Set Computer (RISC) principles, this ISA started as a project for computer architecture research and education, but is aiming at becoming a standard for industrial implementations [1, cap 1]. RISC-V is now increasingly drawing the attention of both industry and academia, because it offers the possibility to ease the design of processors from the high costs of compiler support, without requiring to resort to expensive commercial ISAs.

As stated in the RISC-V Instruction Set Manual one of the objectives of the RISC-V project is to "be used as stable software development target" [1, cap 9].

[1] https://riscv.org/

For this reason the standard defines fixed base integer (*"I"*) ISAs in 32 and 64 bits version plus some optional standard general purpose extensions.

The base integer instruction set, in both in its 32 and 64 bits variant, was devised in order to include a small number of instructions and reduce the complexity of the hardware needed for a minimal implementation. Even if the base ISA is small, it was also designed to be a reasonable target for compilers and software development.

The extensions defined by the standard enhance the capabilities provided by the base ISA with the most commonly used instructions, such as the "F" extension for single precision floating point instructions or the "A" extension for the atomic instructions.

### 1.1.1 Instruction formats

RISC-V base integer ISA defines four basic instruction formats called *R-type*, *I-type*, *S-type* and *U-type* [1] that are shown in 1.1.

These instruction format were defined so that the registers fields for both the sources (*rs1* and *rs2*) and the destination (*rd*) are kept int the same position for all the formats, in order to simplify the decoding hardware. For the same reason the immediates were placed towards the leftmost significant bits and the sign bit position is always in the bit 31 of the instruction.

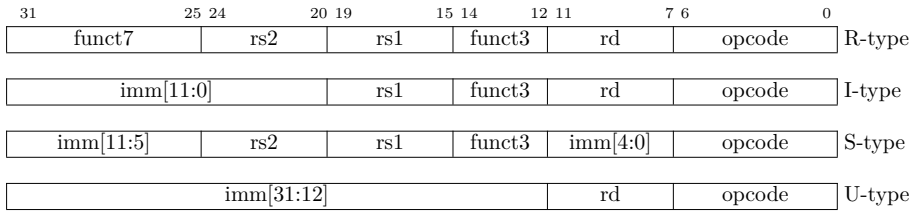


Figure 1.1. RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[$x$]) in the immediate value being produced . [1]

### 1.1.2 ISA extensions

One of the design goals of RISC-V is that of providing support for extensions and customisation [1]. For this purpose the instruction set is organised in a small integer base ("I") plus the optional extensions, among them some of the most common and useful are:

- *"M"*: standard extension for integer multiplication and division instructions.
- *"A"*: standard extension for atomic instructions.
- *"F"*: standard extension for single-precision floating point instructions.
- *"D"*: standard extension for double-precision floating point instructions.
- *"C"*: standard extension for compressed instructions.

The ensemble of *"IMAFD"* extension is indicated with *"G"* and the resulting ISAs are called *RV32G* for the 32-bit version and *RV64G* for the 64-bit one.

On top of the extensions defined by the standard, the ISA also enables the presence of non standard extensions. There are many ways to extend the RISC-V ISA that are extensively described in [1, cap 10].

Among all the possibilities, one of the simplest and the one of interest for this work, is the use of four opcodes, in the 32-bit instruction format, that are reserved for custom extensions. The first two of these custom opcodes, denoted as *custom-0* and *custom-1*, will not be used by future standard extensions, while the opcodes marked as *custom-2* and *custom-3* are reserved for future use by the 128 bit ISA extension.

These four custom opcodes, shown in the table 1.1 are the ones used by the Rocket core processor in the RoCC interface.

| inst[4:2] / inst[6:5] | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 ($> 32b$) |
|---|---|---|---|---|---|---|---|---|
| 00 | LOAD | LOAD-FP | *custom-0* | MISC-MEM | OP-IMM | AUIPC | OP-IMM-32 | $48b$ |
| 01 | STORE | STORE-FP | *custom-1* | AMO | OP | LUI | OP-32 | $64b$ |
| 10 | MADD | MSUB | NMSUB | NMADD | OP-FP | *reserved* | *custom-2/rv128* | $48b$ |
| 11 | BRANCH | JALR | *reserved* | JAL | SYSTEM | *reserved* | *custom-3/rv128* | $\geq 80b$ |

Table 1.1. RISC-V base opcode map, inst[1:0]=11, from the RISC-V ISA maual [1]

## 1.2 Rocket Chip SoC generator and the Rocket Core

This section introduces the Rocket Chip SoC generator and the Rocket Core, two of the main design tools and components around which this work was developed.

These two components are inherently linked together, since the Rocket Chip generator is used to configure and generate the Rocket Core. They are part of the same framework that allows the designers of SoCs to rapidly come up with design architectures.

### 1.2.1 Rocketchip SoC generator

Rocket Chip is an open source SoC generator designed by the Berkeley Architecture Research (BAR) group of the University of California Berkeley (UCB). It is a tool that emits synthesizable RTL, devised to enable designers to build and customise their own SoCs based around the RISC-V ISA [2]. This project is developed in the *Chisel* language[2], a dialect of Scala designed for hardware construction and generation.

The SoC generator allow the composition of modular designs by using the parametrisation features of the *Chisel* language. It offers the possibility of tuning and personalising a lot of configuration parameters of the design, including the support for different standard extension of the ISA, in order to generate different cores able to suit the need of the designer. The project itself can be also viewed as a collection and a library of configurable components for composing SoCs.

The basic design flow allows the integration of new hardware component described in Chisel, the creation of new custom configurations, the compilation of the Chisel/Scala sources to generate C++ models for cycle accurate simulations and the generation of the synthesizable Verilog RTL sources for pushing the design in the standard industry CAD tools.

### 1.2.2 Rocket core

Rocket core is both a processor generator and a library of processor components. As a generator it is able to produce a family of processor core designs, with different configuration parameters. The generated processors have the classical five stage in-order scalar pipeline and can implement the base integer 32-bit RISC-V ISA as well as the 64-bit one [2].

The cores have a Memory Management Unit (MMU) with optional paged virtual memory support, a configurable non blocking private data cache and a front-end with a configurable branch predictor. The generator exposes numerous parameters for configuring the core, among them there is the possibility to add the support for some optional standard ISA extensions (M, A, F, D) and determining the size of the caches [2].

For this study the core was configured implementing the default *RV32G* instruction set configuration, with the addition of the Rocket Custom Coprocessor Interface (RoCC).

[2] https:chisel.eecs.berkeley.edu

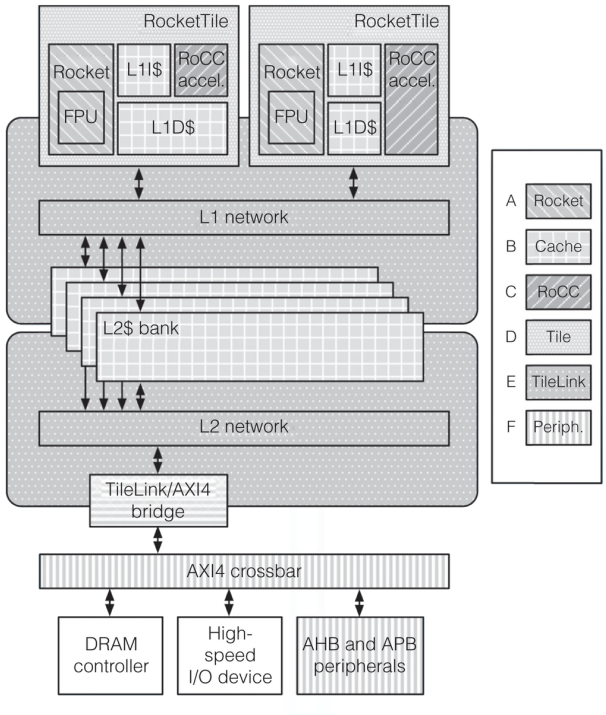


Figure 1.2. Scheme and sub components of the *Rochet Chip SoC generator*, from [2].

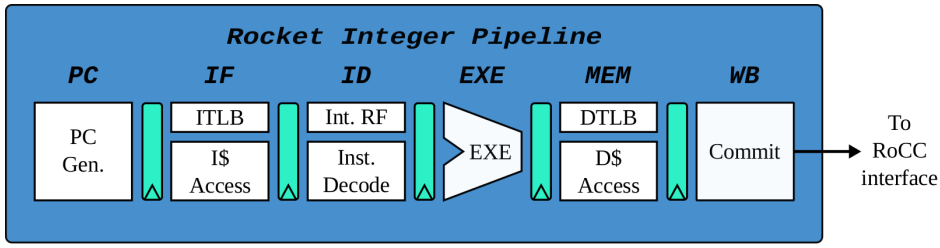


Figure 1.3. Rocket core pipeline

# Chapter 2

# The RoCC coprocessor interface

This chapter describes the main details of the RoCC coprocessor interface and presents an analysis of the latency using the model of *read/write* operations for the data exchange between the core and the accelerator. A *load/store* instruction model is introduced for the data transfers between the coprocessor and the memory. The first section presents the main sets of signals and sub-interfaces composing the RoCC interface, focusing mainly on the ones concerning the communication with the core and the first level data cache. The second section introduces the format of the instructions describing how they are used for the following analysis. The third section describes the method and the latency study performed to evaluate the performance and overheads of the data transfers between the core and the coprocessor. In the end the last section presents a similar kind of analysis describing the use and the performance of the memory sub-interfaces.

## 2.1 RoCC interface overview

The Rocket Custom Coprocessor Interface (RoCC) is an interface designed in order to extend the Rocket Core and allow easy decoupled communications between the core and the attached coprocessors [2]. The RoCC interface is divided in sub-interfaces each creating directional links to connect the accelerators with other parts of the SoC. In particular the *cmd* sub-interface connects the core with the accelerator, as shown in Figure 2.1, and is used to send commands. With these commands the core can also request data to the coprocessor, this data can be sent by the coprocessor to the core through the *resp* (response) interface.

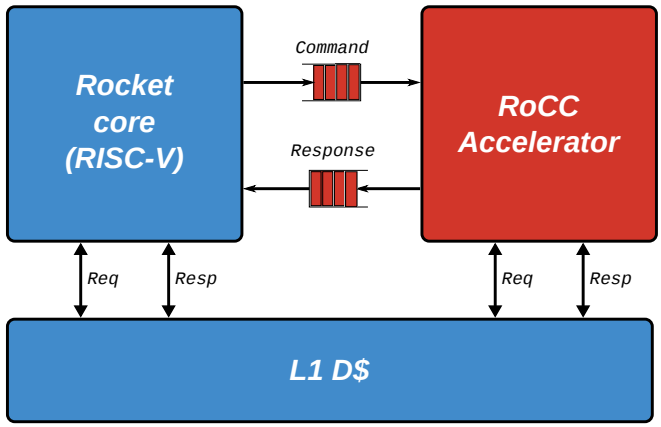


Figure 2.1. Rocket Core and a coprocessor connected with command and response interfaces.

In order to allow the coprocessor to access the memory the RoCC interface also provides the *mem_req* (request) and *mem_resp* (response) direct links to the data cache.

Other than these four channels the RoCC interface provides some more sub-interfaces to allow advance functionality, in particular it is possible to connect an accelerator with the FPU, to share the page table walker and to have a direct link with the outer memory system [2]. These extra sub-interfaces are part of the so called *extended RoCC interface*.

The interface also provides some more status signals and an interrupt line that can be used for syncronising with the core or for signalling errors.

In the context of this document and for the following analysis only the base RoCC interface composed of *cmd*, *resp*, *mem_req* and *mem_resp* sub-interfaces will be considered.

### 2.1.1 RoCC command and response interfaces

As previously said the RoCC interface defines two sub interfaces for the data exchange between the accelerator and the core. The command interface is used to send the instructions and the corresponding data to the coprocessor, while the response interface is used by the coprocessor to send the results to the integer register file.

Both the command and the response ports are based on the *Decoupled* interface available in Chisel. This type of connection is based on a FIFO like *ready/valid* protocol in which the sender drives the *valid* signal and the data and

waits for the receiver to raise the *ready* signal, the transfer is considered accepted if both valid and ready are *high* on the same clock cycle.

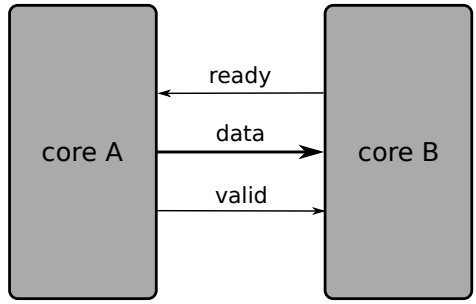


Figure 2.2. Two cores connected through the *Decoupled* interface

Based on this type of connection the core can send commands to the accelerator by rising the valid signal whenever there is a legal custom instruction that reaches the *write-back* stage. The data bus of the command interface is composed of the following signals:

- **inst** the full 32-bit instruction
- **rs1** a 32-bit (or 64 depending on the *XLen* parameter) data bus for the content of the integer register addressed by the *rs1* field of the instruction
- **rs2** a 32-bit data bus holding the value of the integer register addressed by the *rs2* field of the instruction

The *inst* signal is further subdivided in the fields defined by the RoCC custom instruction. The format of the RoCC instructions follows the *R-type* format shown in 1.1, but the opcode section is bound to assume one of the four possible custom opcodes values, while the *funct3* part is divided into three bit fields.

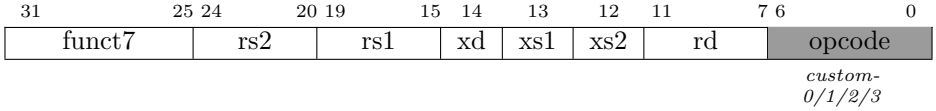


Figure 2.3. RoCC instruction format

These bit fields are quite important for the correct use of the RoCC interface, they are in fact used as a kind of valid bit for the registers specifiers in the instructions:

- **xd** bit is set when *inst_rd* is a valid destination register: the core wants to receive data in the destination register pointed by *inst_rd*.

- **xs1** bit is set when *inst_rs1* is a valid source register: the core is sending the content of the first source register (*inst_rs1*) in the *rs1* data bus.

- **xs1** bit is set when *inst_rs2* is a valid source register: the core is sending the content of the second source register (*inst_rs2*) in the *rs2* data bus.

The values assumed by these bits heavily influences the behaviour of the pipeline, in particular it is frequent to cause stalls when these bits are set, this applies in particular for *inst_xd*.

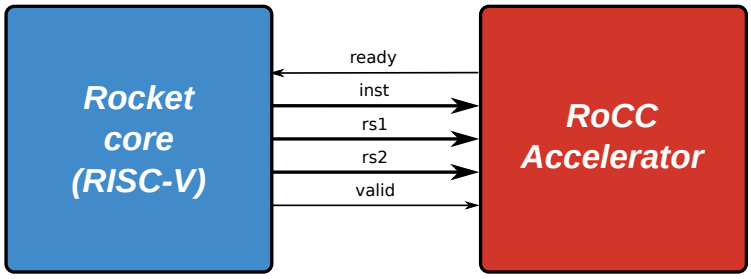


Figure 2.4. The *cmd* interface signals

When a custom instruction with the *inst_xd* bit set arrives, the core will expect to receive, at some point, a result to be stored in the register pointed by *inst_rd*. This means that if a successive instruction uses that register before the value is produced and stored, the core will stall the pipeline to wait the data from the coprocessor.

In order to serve these requests, the accelerator have to use the *resp* sub-interface, which is also a *Decoupled* (ready/valid) interface. In this case the coprocessor drives the valid signal and the "data" bus, while the core only drives the ready signal. The data information includes the following buses:

- **rd** the five bits field specifying the destination register of the response, must be the same received with the command.

- **data** a 32 or 64-bit bus with the data content to be written in the *rd* register.

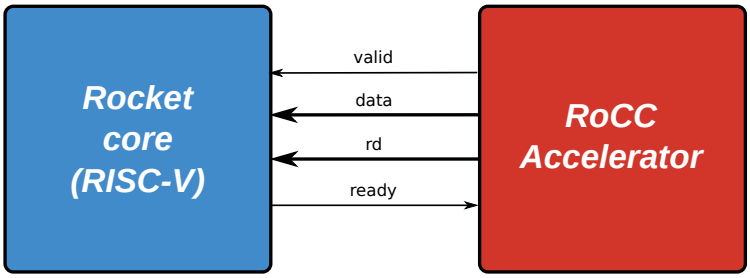


Figure 2.5. The *resp* interface signals

### 2.1.2 Memory request and response interfaces

In order to allow the accelerator to have direct access to the first level data cache, the RoCC interface specifies two channels for the memory requests *mem_req* and for the responses *mem_resp*.

#### mem_req sub-interface

When an accelerator wants to issue a load or store operations to the memory it can use the *mem_req* sub-interface. As for *cmd* or *resp* also the *mem_req* link is based upon the *Decoupled* interface. Aside from the *ready* and *valid* signals this connection provides in the data section a lot of signals, some of which where not used for this work, in particular the most important data signals of this sub-interface are:

- **addr** a 32 (for RV32) or 40-bit (for RV64) bus carrying the address for the memory access.
- **tag** 8-bit bus used to uniquely identify each request.
- **cmd** 5-bit bus carrying the memory operation code ($0000_2$ = load, $0001_2$ = store)
- **typ** 3-bit bus that specifies the width of the of the transfer operation ($000_2$ = 8-bit, $001_2$ = 16-bit, $010_2$ = 32-bit and 011 = 64 bit).
- **phys** 1 bit signal asserted if a physical address is used or zeroed if the address is virtual and needs a translation.
- **data** 32 or 64-bit bus used for sending the data in case of store operations.

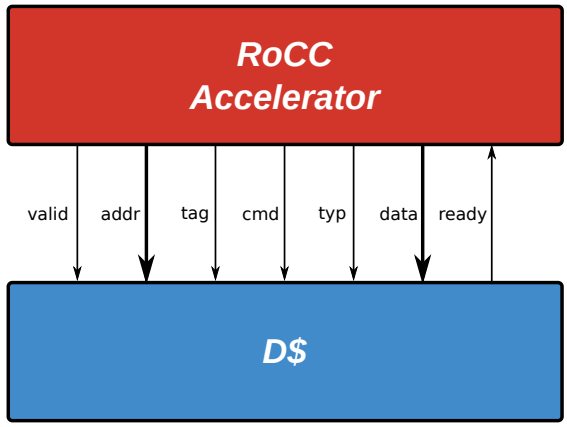


Figure 2.6. *mem_req* sub-interface main signals

### mem_resp sub-interface

When the response from the cache is ready it is sent back to the accelerator through the *mem_resp* sub-interface. This link present a substantial difference with respect to the previously discussed ones, in fact it is not based on the *Decoupled* interface. This is because the coprocessor cannot keep the cache waiting, so it must accept the response without the possibility of postponing the transaction. In fact the *mem_resp* sub-interface does not present a *ready* signal, but is instead based on a simpler *"valid"* interface. This means that the coprocessor must accept every memory response as soon as the valid line is high. With this interface the sender (the cache) drives the *data* and the *valid* signals and the receiver (the accelerator) just need to check that the valid line is high and accept the incoming data.

The data content of this interface is also populated with a lot of signals, but the main ones are mostly the same as the *mem_req*:

- **addr** a 32 (for RV32) or 40-bit (for RV64) bus carrying the load/store address.
- **tag** 8-bit bus used to distinguish between responses to multiple requests.
- **cmd** 5-bit bus carrying the memory operation code ($0000_2$ = load, $0001_2$ = store)
- **typ** 3-bit bus that specifies the width of the of the response data ($000_2$ = 8-bit, $001_2$ = 16-bit, $010_2$ = 32-bit and 011 = 64 bit).

- **data** 32 or 64-bit bus carries the data response for a load operation.

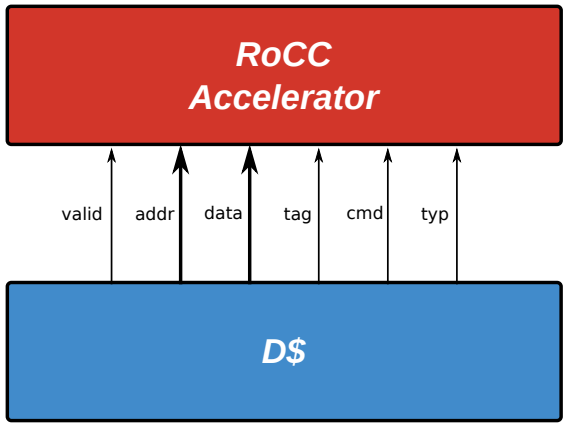


Figure 2.7. *mem_resp* sub-interface main signals

As it was said earlier the memory response channel does not provides a way to control the flow of responses, moreover this sub-interface does not guarantee that the responses will arrive in order. This means that the coprocessor must deal with the control of the data flow and must adopt some strategy to take care of possible out-of-order responses.

### 2.1.3 The extended RoCC interface

The *extended RoCC interface* includes a set of optional extension channels to allow the development of coprocessors with more advanced features. In particular the additional sub-interfaces are:

- **aUTL** arbitrated Uncached Tile Link used for communications whit the outer memory system, is divided into an *acquire* and a *grant* channels, both based on the *Decoupled* interface (ready/valid protocol).
- **FPU** used for allowing communications with the floating point unit, it is divided into *request* and *response* and both channels are based on *Decoupled* links.
- **PTW** used for coprocessors that need to talk with the Page Table Walker (PTW), this sub-interface is also based on *request* and *response* Decoupled links, but it also provides additional sets of signals, moreover it is possible to generate coprocessors with more than one of this sub-interfaces based on how many PTW channels are needed.

Even if some of these extensions might be of interest for the development of accelerators, in particular the aUTL, this work only focuses on the use of the base RoCC interface, that should be able to suit the needs of the majority of the coprocessors applications.

## 2.2 Custom instructions

This section briefly introduces the instructions and the programming model used for the successive analysis of the RoCC interface. As it was described before this interface make use of the four custom opcodes, available in the base 32-bit length instruction space. On top of this the format of the instruction is fixed by the interface and, since the use of some bit fields of the RoCC instruction heavily impact the behaviour of the pipeline, it was decided to follow this format but adopting a particular addressing convention. Moreover to simplify the coprocessor model for the interactions and data transfer with the core and with the data cache only four operations are defined: read/write and load/store.

### 2.2.1 The addressing mode

The three bit fields in the RoCC instructions *xd*, *xs1* and *xs2* are used to validate the source or destination registers on the core side, for this reason it was decided, as a convention, that when one of these bits is zero the corresponding register field refers to an internal register of the accelerator. This basically means that we can have custom instructions with the *xs1* and/or *xs2* bits sets to read values form the integer register file of the core and send it to the accelerator, while an instruction that zeroes those fields is addressing the internal registers of the accelerator. As it can be seen in the table 2.1, when one of the three bit fields is set to one, the corresponding register field in the instruction is referring to a Core register. When a bit is zero, the register field of the instruction is used to address an internal register of the RoCC accelerator. The configurations using only *inst_rs2* as source (with just *xs2* at one) cannot be decoded by the Rocket Core.

Other than this the interface also allows an additional mode of addressing: it is possible to use the value of one or both the *rs1* and *rs2* data bus, of the *cmd* sub-interface, as addresses for the internal registers of the accelerator. Note that *inst_rs1* indicates the 5-bit field of the instruction for the register specifier, while *rs1* is used to indicate the 32-bit (or 64-bit) data bus carrying the content of the register in the *cmd* sub-interface.

Using the data arriving from the core for the internal registers addressing grants two main advantages: the first one is the possibility of using register banks wider than 32 registers, since in principles it is possible to address up to

| xd | xs1 | xs2 | inst_rd | inst_rs1 | inst_rs2 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | RoCC | RoCC | RoCC |
| 0 | 0 | 1 | not allowed | | |
| 0 | 1 | 0 | RoCC | Core | RoCC |
| 0 | 1 | 1 | RoCC | Core | Core |
| 1 | 0 | 0 | Core | RoCC | RoCC |
| 1 | 0 | 1 | not allowed | | |
| 1 | 1 | 0 | Core | Core | RoCC |
| 1 | 1 | 1 | Core | Core | Core |

Table 2.1. The convention for the use of the *xd*, *xs1* and *xs2* bits.

$2^{32}$ registers. The second benefit is that, since the content of the *rs1* and *rs2* busses comes form the integer registers, it is possible at the code level to address the internal registers of the accelerator by using normal integer variables.

From this results that two register addressing modes are possible. One using the registers fields in the instruction, with which only the first 32 registers can be accessed. The second that uses the *rs1* and *rs2* data bus content for the internal addressing, which allows basically an infinite register address space and the possibility to use variables in the high level code for the addressing. These two addressing modes will be called respectively instruction based addressing and data based addressing.

### 2.2.2 Instructions

Considering the *read/write - load/store* model and the two addressing modes it is possible to define our basic instructions. Since each RoCC coprocessor is mapped to a different custom opcode the opcode field does not carry information about the instructions. For this reason the *funct7* field is used to specify the operation opcode, while the different configurations of the *xd*, *xs1* and *xs2* bits are used to determine the addressing mode. The instruction defined following this approach are shown in the table 2.2.

## 2.3 Read/write operations between the core and RoCC

This section introduces a latency analysis on the RoCC interface, in particular the analysis focuses on the latency of the read operations. As discussed in the previous section the data movements between the core and the coprocessor are modelled following the read/write model, in particular we consider as if the core

| funct7 | xd | xs1 | xs2 | inst_rd | inst_rs1 | inst_rs2 | rs1 | rs2 | operation |
|---|---|---|---|---|---|---|---|---|---|
| **read** | 1 | 0 | 0 | Core | Acc | - | data1 | - | C[inst_rd] ← A[inst_rs1] |
| **read** | 1 | 1 | 0 | Core | Core | - | data1 | - | C[inst_rd] ← A[data1] |
| **write** | 0 | 1 | 0 | Acc | Core | - | data1 | - | A[inst_rd] ← data1 |
| **write** | 0 | 1 | 1 | Acc | Core | Core | data1 | data2 | A[data2] ← data1 |
| **load** | 0 | 1 | 0 | Acc | Core | - | data1 | - | A[inst_rd] ← M[data1] |
| **load** | 0 | 1 | 1 | - | Core | Core | data1 | data2 | A[data2] ← M[data1] |
| **store** | 0 | 1 | 0 | - | Core | Acc | data1 | - | M[data1] ← A[inst_rs2] |
| **store** | 0 | 1 | 1 | - | Core | Core | data1 | data2 | M[data1] ← A[data2] |

Table 2.2. Instructions defined for the RoCC coprocessors. "C[x]" indicates an access to the x-th integer register, "A[i]" indicates an access to the i-th register of the accelerator, "M[x]" is used for a memory access at the address x.

is performing read or write operations, according to this convention the following pseudo-instructions are defined:

- *rocc_read rd, cps*: the core reads the data arriving from the coprocessor register *cpx* and write it into the destination register *rd*.

- *rocc_write cpd, rs*: the data in the core register *rs* is written in the accelerator register *cpd*.

The following latency study was performed in absence of cache misses, considering the updates of the Instruction Register (IR) and of the Program Counter (PC) in the Write Back (WB) stage for counting the clock cycles for each instruction.

### 2.3.1 Latency study

This study was performed using a RoCC coprocessor capable of responding in a combinational way to the read operations, like in a normal read from a basic register file. This means that the coprocessor itself is not introducing any additional latency, so that the measured latency is only dependent on the interface.

The analysis is focused mainly on the performance of read operations, this is because the coprocessor and the core are decoupled. After the *rocc_write* instruction is sent there is no result to wait, so from the core point of view the instruction has been executed correctly. Moreover it is reasonable to suppose that write operations only requires one cycle as in the case of writes into a normal register.

This analysis covers two main scenarios: the read of a data from the coprocessor and its subsequent use by a normal RISC-V instruction and a burst of read operations followed by the usage of the read values.

### read and use operations

This scenario is meant to cover the cases when, after some computation, a *rocc_read* is performed to retrieve some result data that is needed by the successive instructions. In particular the test used a *"rocc_read rd, cpx"* instruction, that read the *cpx* register of the accelerator and stores the result in the core's register *rd*, and a *"use rx, rd"* instruction, in particular a *"addi rx, rd, 1"*, that exploits the data stored in *rd*.

The read/use instructions were interleaved by a varying number of independent instructions, *"nop"* were used for this purpose. The idea was to try to hide the latency of the interface, by performing some useful computation independent from the result of the *rocc_read*. This technique is a common optimisation practice also adopted by many compilers, but, as it can be seen in the Figure 2.8, no matter how many independent instructions are inserted between the *rocc_read* and the *use*, reading from the RoCC interface always results in a big latency penalty.

| | | | |
|---|---|---|---|
| *rocc_read* | rd, cpx | ; rd ← RoCC[cpx] | **5** cycle |
| *addi* | rx, rd, 1 | ; rx ← rd + 1 | 1 cycle |
| | | | |
| *rocc_read* | rd, cpx | ; rd ← RoCC[cpx] | 1 cycle |
| nop | | ; | **4** cycle |
| *addi* | rx, rd, 1 | ; rx ← rd + 1 | 1 cycle |
| | | | |
| *rocc_read* | rd, cpx | ; rd ← RoCC[cpx] | 1 cycle |
| nop | | ; | 1 cycle |
| ... | | | |
| nop | | ; | **4** cycle |
| *addi* | rx, rd, 1 | ; rx ← rd + 1 | 1 cycle |

Figure 2.8. Read/use scenario test code and corresponding execution time in clock cycles.

### burst of read operations

Another scenario that has been tested is the use of multiple consecutive *rocc_read* operations. These tests showed even more the limitations of the *RoCC* interface, in fact the read operations not only imposes frequent stall but it also starts to show throughput limitations when more than four consecutive *rocc_read* instructions are issued.

As it is shown in the plot 2.9, in order to minimise the penalty the best number of consecutive *rocc_read* instruction must be two or three. As it was said earlier

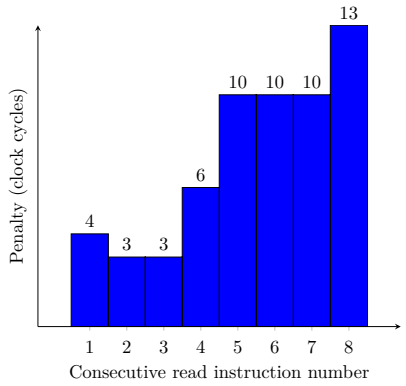


Figure 2.9. Latency penalty with different numbers of consecutive reads.

```
...
rocc_read   rd1,   cpx1        ; 1 cycle
rocc_read   rd2,   cpx2        ; 1 cycle
rocc_read   rd3,   cpx3        ; 4 cycle
use         rd1,   rd1,   1    ; 1 cycle
use         rd2,   rd2,   1    ; 1 cycle
use         rd3,   rd3,   1    ; 1 cycle
...
```

Figure 2.10. Test-case example with three consecutive reads.

more than four reads caused throughput issues, while a single read suffers from an additional cycle of stall.

### 2.3.2 Improving latency

This section analyses the main causes for the pour latency results of the RoCC inteface and tries to provide some workarounds to improve the performance of the interface.

The main reasons behind the relatively high latency overhead introduced by the interface are to be researched in the position of the interface in the pipeline. In fact the Rocket core forwards the custom instructions through all the stages of the pipeline, down to the WB stage, at this point the instruction is sent to the RoCC command router. The RoCC command router is a configurable component component that buffers the *custom* instructions and forwards them to the right coprocessor, based on the opcode. The presence of this additional component introduces an additional time barrier and so an additional clock cycle of latency before the instruction can reach the coprocessor. This choice can be explained because it allows simpler accelerators that do not need to deal with exceptions. In fact when a *custom* instruction reaches the WB stage it is guaranteed that the previous instructions were correctly executed and that they did not caused any exception. With an interface placed in the Execute (EXE) stage, a couple of clock cycles could have been gained, but the coprocessors would have been more complex in order to deal with the exceptions. Also it must be remembered that the RoCC interface was devised for enabling communications with decoupled

coprocessors [2]. All of these reasons can explain why the interface is in such a late position in the pipeline.

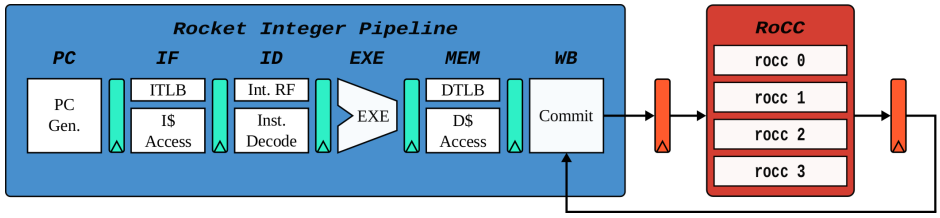


Figure 2.11. Position of the RoCC interface in the pipeline

The other major cause of penalty, that limits the throughput of *rocc_read* operations, can be found in the way the core handles the write-back of data coming from the interface. In particular the core gives priority for the write in the Register File (RF) to the instruction that is currently occupying the WB stage. This means that the data arriving from the *RoCC* interface is written in the RF only when the WB stage is occupied by a "non writing" instruction, like for example branches. This implies that every instruction that writes on the RF (basically any arithmetic instruction like *"add"*, *"or"* etc..) preempts the writing of data coming from the RoCC *resp* interface. The pending response is going to be accepted only when an instruction leaves the WB stage free, or when the data coming from the coprocessor is needed by an instruction in the EXE stage, causing a stall. This kind of behaviour also applies to the *"nop"* instruction, witch is implemented as an *"addi x0, x0, 0"*, as defined by the RISC-V standard [1].

At the $\mu$architectural level every instruction has a bit in the control-word, called the *wxd* bit, that is asserted in any instruction that needs to write in the RF. In the WB stage a check on the *wxd* bit is performed and the *ready* signal of the RoCC *resp* interface depends on that bit being at zero. For this reason as long as there is a valid instruction with the *wxd* bit at one, in the WB stage the writing form the *resp* interface is preempted.

A possible way to avoid paying a stall for every *rocc_read* consist in masking the latency of the interface with some useful instructions with an additional "useless" instruction used as a kind of free write-back slot. In order to apply this strategy the useless instruction must be one with the *wxd* bit at zero and must not alter the internal state of the core. Basically a *"nop"* instruction that does not occupy the WB slot is needed.

To provide this capability two possible strategy are presented, one that implies

a slight modification to the core and one that uses a kind of *"nop"* defined in a different way.

### Modification of the NOP operation

As it was said earlier the *ready* signal of the RoCC *resp* interface depends on a check on the value of the *wxd* bit, in particular the signal is computed as follows:

$$wb\_wxd = wb\_reg\_valid \;\&\&\; wb\_ctrl.wxd$$

$$rocc.resp.ready = !wb\_wxd$$

Where the *wb_reg_valid* is the valid flag for the instruction in the WB stage, *wb_ctrl.wxd* is the *wxd* bit of the control word for the instruction in the WB stage and *wb_wxd* is the final *wxd* bit that is asserted when there is a valid write in the RF. The symbol "&&" is used to indicate the logical *and*, while "!" is used for the logical negation (*not*). The *ready* signal for the *resp* interface is asserted only when *wb_wxd* is zero.

By introducing an additional check on the destination register not being zero, in the computation of *wb_wxd* it is possible to grant the write back slot to the RoCC response for all the instructions that, like the *"nop"*, have zero as destination register:

$$wb\_wxd = wb\_reg\_valid \;\&\&\; wb\_ctrl.wxd \;\&\&\; (wb\_waddr \neq 0)$$

By introducing this modification in the core it is possible to exploit the following type of strategy for hiding the latency of read operations:

```
...
rocc_read    rd, cpx     ; rd ← Coproc[cpx]          1 cycle
instr1                   ; usefull computation       1 cycle
instr2                   ;                           1 cycle
nop                      ; RoCC write-back slot      1 cycle
instr3                   ;                           1 cycle
instr4                   ;                           1 cycle
instr5                   ;                           1 cycle
addi         rx, rd, 1   ; use rd                    1 cycle
...
```

Figure 2.12. Masking the latency of a read operation by inserting some independent instructions with a free cycle for the write-back of the RoCC interface.

### Using useless branch to mask the latency

By substituting the *nop* instruction with a useless branch never taken, it is also possible to avoid any modification of the core. Using this method allows to exploit the same code sequence, but it uses a non standard definition of the *nop* instruction.

```
...
rocc_read     rd, cpx      ; rd ← Coproc[cpx]        1 cycle
instr1                     ; usefull computation     1 cycle
instr2                     ;                         1 cycle
bnez          zero, addr   ; RoCC write-back slot    1 cycle
instr3                     ;                         1 cycle
instr4                     ;                         1 cycle
instr5                     ;                         1 cycle
addi          rx, rd, 1    ; use rd                  1 cycle
...
```

Figure 2.13. Use of a branch never taken as write cycle for the RoCC interface.

The approach shown in the figures 2.12 and 2.13, allows to pay a single cycle penalty when reading from the RoCC *resp* interface. Even if it may give a good performance improvement, this remains a very situational optimisation, which would be difficult to apply to a compiler. The exploited sequence must be respected but in general it may be impossible to find the exact number of independent instructions. Moreover the sequence is strictly dependent on the latency of the coprocessor, so it would be almost impossible to rely on this approach without a knowing the details of the accelerator.

## 2.4 Loads and stores between RoCC and cache memory

This section describes the use of the memory request and response sub-interfaces to allow a coprocessor to directly fetch data from the L1 data cache. Following the model used for the communications with the core, for the data transfers between the coprocessor and the data cache the following pseudo-instructions are defined:

- *rocc_load cpd, rs*: the coprocessor performs a load operation using the content of the integer register *rs* as memory address and puts the result in its internal register cpd.
- *rocc_store cps, rs*: the coprocessor performs a store operation of the value contained in *cps* using the content of the register *rs* as memory address.

The presence of single memory operation instructions may seem to give little advantages because it is almost equivalent to performing a normal RISC-V load plus a *rocc_write* on the coprocessor. The use of single transfer instructions is also limited because, before launching this kind of instructions, the integer register *rs* should be initialised with the proper address. This basically results in a overhead of at least one additional instruction for each *rocc_load*, for preparing *rs*, this usually means moving the address in the register as it is shown in Figure 2.14. Moreover as the RoCC interface is placed after the WB stage the memory access is delayed by at least two clock cycles (the actual number may depend on the accelerator).

```
...
addi        a4, sp, 24    ; prepare address #1
addi        a5, sp, 16    ; prepare address #2
addi        a5, sp, 8     ; prepare address #3
mv          a6, sp        ; prepare address #4
rocc_load   cpd1, a4      ; Coproc[cpx] ← Mem[a4]
rocc_load   cpd2, a5      ; Coproc[cpx] ← Mem[a5]
rocc_load   cpd3, a6      ; Coproc[cpx] ← Mem[a6]
rocc_load   cpd4, a7      ; Coproc[cpx] ← Mem[a7]
...
```

Figure 2.14. Sequence of *rocc_load* with address preparation instructions.

However the memory interface is probably the best way to move a big amount of data in and out of the coprocessor.

Another possibility could be to implement burst transfers, with a single instruction or with a sequence of programming instructions. In order to perform a burst transfer the coprocessor needs to know at least: the base address, the size of the transfer and the destination/source register (supposing that the size of the data is fixed). Instructions for fixed size bursts (8 words, 16 words etc..) can be implemented to be able to send all the required information with a single instruction. On the other hand a more flexible approach could use fully programmable burst, this choice would lead to the use of multiple write instructions to send all the required information. This last option could be effective only for long transfers, because of the overhead of the programming instructions.

In general the use of burst instructions is heavily application dependent and not general enough for implementing them in every coprocessor.

### 2.4.1 Latency study

Independently if the memory interface is used by single or burst load/store instructions, or by internal operations of the coprocessor, the behaviour of the *req* and *resp* channel is basically the same. The following analysis considers the behaviour with a single memory operation and with consecutive memory operations. The analysis does not consider the presence of cache misses, this is because it would depend on many factors, for example the memory accesses of the program running on the core. Moreover it is difficult to consider cache misses in the analysis because the caches are completely configurable in size, organisation, coherency protocol and the data cache can be also substituted by a scratchpad memory.

As it was said in the description of the interface the *req* channel is a *Decoupled* link. This means that each request is considered accepted when both the *ready* signal from the cache and the *valid* signal from the coprocessor are high in the same clock cycle.

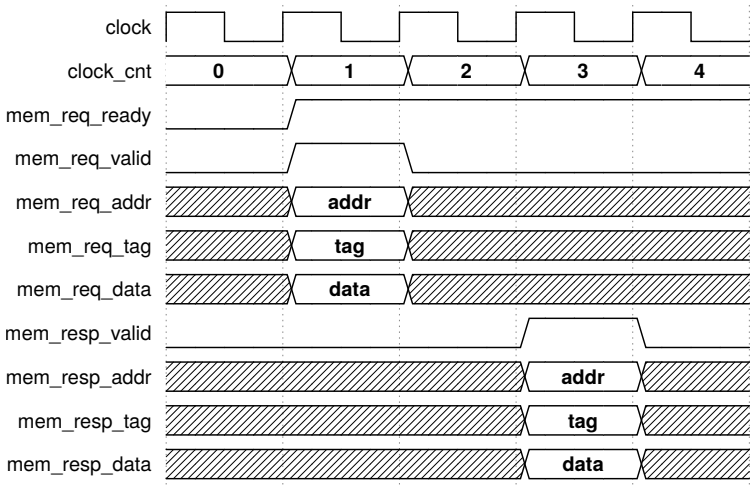


Figure 2.15. Single memory request and response waveform.

The Figure 2.15 shows a timing diagram for the main signals of the request and response sub-interfaces for a single memory operation. As it can be seen, the response arrives in the second clock cycle after the request is fired (*ready* and *valid* both high in the same clock cycle).

The Figure 2.16 shows the behaviour of the memory interface when the coprocessor tries to perform multiple consecutive memory access.

As it can be seen the memory is able to accept two consecutive requests,

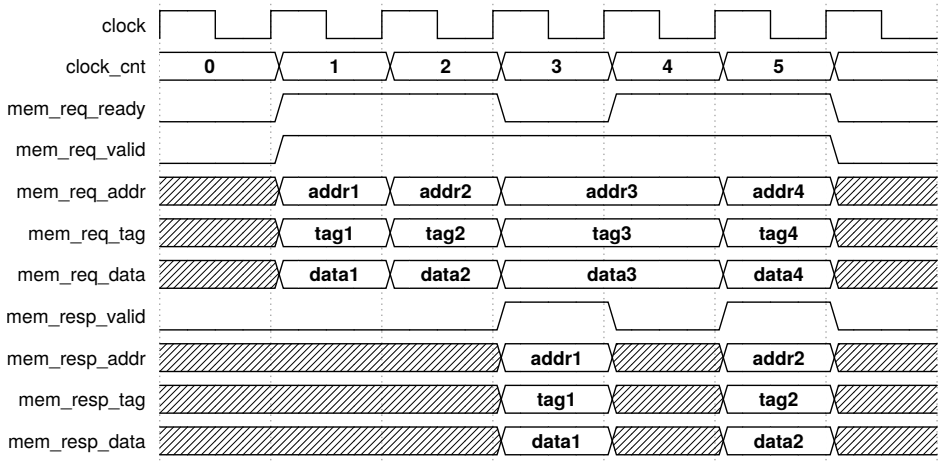


Figure 2.16. Multiple consecutive memory request waveform.

then it needs a cycle of wait before the next two requests can be accepted. The resulting maximum throughput of the interface is 50% and the latency is 2 clock cycles.

# Chapter 3

# Case study: the Blockcipher coprocessor

This chapter presents the design of a cyptographic accelerator, starting from a block-cipher module internally developed at CEA. This core is able to perform both AES and PRESENT encryptions and is optimized for low power and low energy[4] but was not designed to be part of a coprocessor system. The first section provides an overview of the block-cipher core, describing its interface, the mode of operation and the input/output protocols and timings. The second section describes the architecture of the design, outlining the register file, the state machines and their interactions. It present an extension of the cryptographic module, that enhance the functionality and improve the coprocessor performance. The third section discusses the programming model and the instruction set defined for this design. The last section presents a performance analysis that considers the two main mode of operation of the accelerator.

## 3.1 Overview of the Block-cipher core

This section briefly introduces the concept of block-cipher and the block-cipher core described in [3] and [4]. A block-cipher is a cipher algorithm that takes a block of plain-text and use it to produce a block of cipher-text of equal length [5]. This kind of algorithms are very widely used, and hardware implementations of these algorithms are often exploited to achieve greater performances and lower power consumption.

In particular the block-cipher core used for this study implements the Advanced

Encryption Standard (AES) and PRESENT algorithms. AES, also known as Rijndael, is probably the most widespread block-cipher and was published as a standard by the National Institute of Standards and Technology (NIST) in 2011 [5]. AES uses a block size of 128-bit and supports key sizes of 128-bit, 192-bit and 256-bit.

PRESENT on the other hand is a newer light-weight block-cipher, devised for being suitable for constrained devices like the ones used in the IoT domain [6]. PRESENT algorithm exploit a 64-bit block size and support 80-bit or 128-bit key sizes.

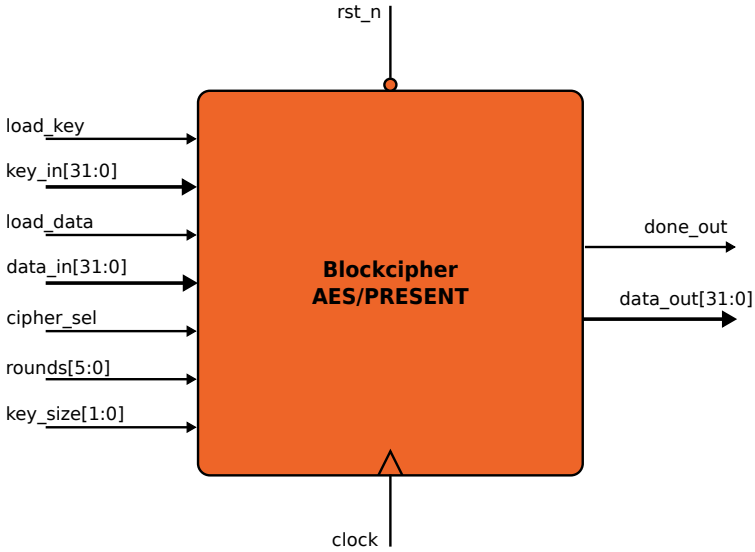


Figure 3.1. Schematic view of the block-cipher interface.

The block-cipher core used in this study was also designed having in mind constrained devices like those of the IoT. The datapath is on 32-bit optimised for ultra low-power and ultra low-energy consumption. It is capable of performing both AES and PRESENT encryption with all the supported key sizes (128-bit, 192-bit and 256-bit for AES and 80-bit and 128-bit for PRESENT).

As it can be seen in the picture 3.1, the block-cipher IP has a common 32-bit interface, with a 32-bit bus for the key and one for the data, and a pair of load enable for key and data chunks. It is possible to configure the cipher (AES or PRESENT), the key size and the number of rounds of the algorithms with dedicated selection signals. The output interface presents a 32-bit bus and a *"done"* signal used to signal a valid data.

The Figure 3.2 shows the main signals involved in the input/output protocol of an AES 128-bit encryption.

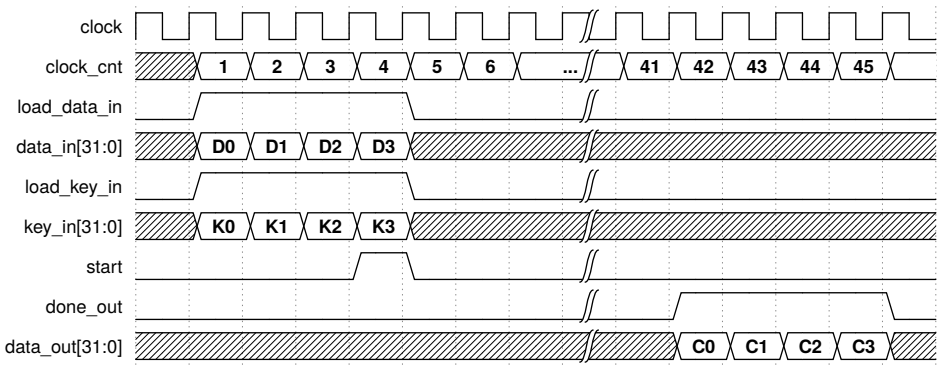


Figure 3.2. Time diagram for the block-cipher IP AES-128 input/output protocol.

## 3.2 Architecture of the Block-cipher coprocessor

This section presents the details of the proposed architecture for embedding the block-cipher IP in a RoCC coprocessor. Overall the architecture presents three major components: the interface module, the computing core and the memory module.

As it can be seen in the Figure 3.3 the interface module is the component in charge for the communication between the processor core and the register bank.

The computing core is based on the block-cipher IP described in the previous section, with some additional components used to obtain new functionality. In particular FIFO queues are used to pipeline the operations and to buffer the loading of new plain-text data and storing of the resulting cipher-text data.

The last block in the diagram is the memory module which is in charge of performing load and store operation to fill and spill the computing core queues. Since both the interface module and the memory module can perform memory accesses, the RoCC memory sub-interfaces are multiplexed between the two.

### 3.2.1 Interface module

The interface module is the component that is connected to the processor with both the *cmd* and *resp* interface. It is able to directly communicate with the L1 data cache by implementing the full RoCC memory sub-interface. Given

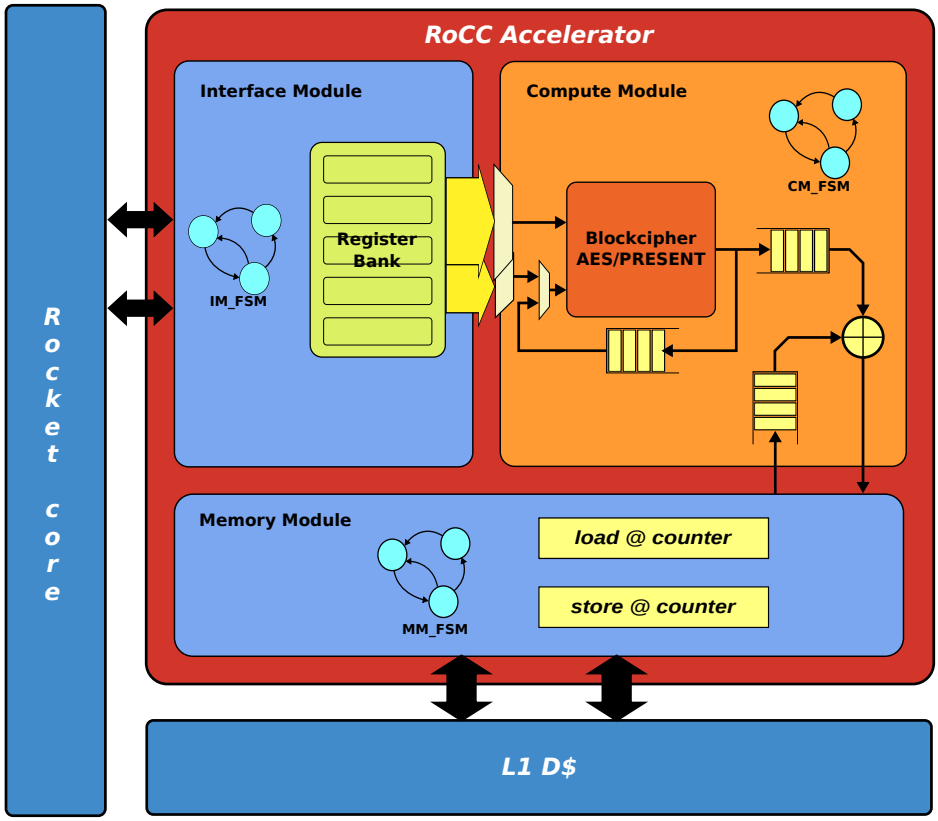


Figure 3.3. High level block diagram of the block-cipher coprocessor.

the programming model of *read/write* and *load/store*, the interface module is basically a bridge for data transfers between the processor and the register bank and the memory and the register bank.

The interface module is in charge of receiving and decoding commands from the processor, and performing flow control by deciding when to accept a new given instruction. The *ready* signal of the RoCC interface is used by the interface module to manage the flow of instructions. To decide whether to rise or keep low this signal, the interface module performs several checks based on the value of the *funct7* field of the RoCC instruction (Chapter 2). The control logic implements the following checks:

- when $funct7 = READ\_OPCODE$:

check if a read operation can be executed.

- when $funct7 = WRITE\_OPCODE$:

  check if a write operation can be executed.

- when $funct7 = LOAD\_OPCODE$ *or* $funct7 = STORE\_OPCODE$

  check if a memory operation can be executed.

Then depending on the type of instruction decoded, the interface module uses the result of this checks to determine the value of the ready signals. These checks depends on several conditions which are reported in the pseudo-code in Figure 3.4.

```
-- Accept a read only if response interface is ready and RF can reply
-- registers above the 15th are read-only so they can always be read
rd_addr_ok  <= (rd_addr > 15) or (mem_mod_ready and encr_ready);
read_ok     <= rocc_resp_ready and cfg_rsp and rd_addr_ok and not(enc_start);

-- Accept a write instr. if nobody is reading or writing
write_ok <= encr_ready and mem_mod_ready and cfg_rsp and not(enc_start);

-- Accept a memory instr. if memory is ready and no one is using it
mem_ok   <= mem_req_ready and encr_ready and mem_mod_ready and cfg_rsp;
```

Figure 3.4. Pseudo-code for the ready check logic.

Thanks to these checks whenever an instruction can be computed is immediately accepted, while instructions that requires some busy resource are kept waiting.

The interface module also implements a Finite State Machine (FSM) for handling the memory operations. In particular the FSM is used to guarantee the correct order of the memory operations, by implementing a wait state for the memory response, as it can be seen in Figure 3.5.

During the two wait states the interface module will keep the *ready* signal of the command interface low, to avoid accepting instructions that could corrupt the state of the coprocessor.

### 3.2.2 Register bank

The register bank was designed with a code-generation tool internally designed by CEA. This tool allows to generate configuration register banks starting from a specification file. The register bank support a bus communication interface on one side and give parallel access on the Intellectual Property (IP) side, so

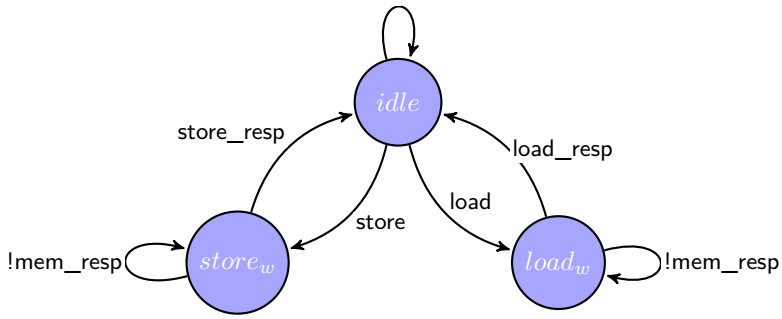


Figure 3.5. Interface module FSM (IM_FSM).

that it can be used to generate configuration registers for peripheral cores. The generated register bank is able to handle conflicting accesses by givin priority to the IP core side and not giving a valid response to the interface request. The tool can be configured to produce an AMBA APB interface or the *cfg* interface.

The *cfg* interface has been chosen for this work because it offers a simpler and faster interface, more suitable for a coprocessor register file. The *cfg* interface is composed of the following signals:

- *addr*: address of the request.
- *wdata*: write data.
- *write*: write enable for the request.
- *req*: request enable.
- *rdata*: read data.
- *rsp*: response valid.
- *error*: error code.

For this design the tool was configured for producing a register bank with a 5-bit address space, the registers were organised as shown in Figure 3.6.

The registers containing the key (from 0x00 to 0x07) and the input data (0x08 to 0x0B) are configured to be readable and writable for both the interface side and the IP side. The addresses from 0x0C to 0X0F are read-only for the CPU and are used to store the result of a single block encryption.

The *msg size* (0x011) is the register used to set up the size of the plain-text for an OFB encryption. The *plain addr* regiseter (0x12) holds the source base

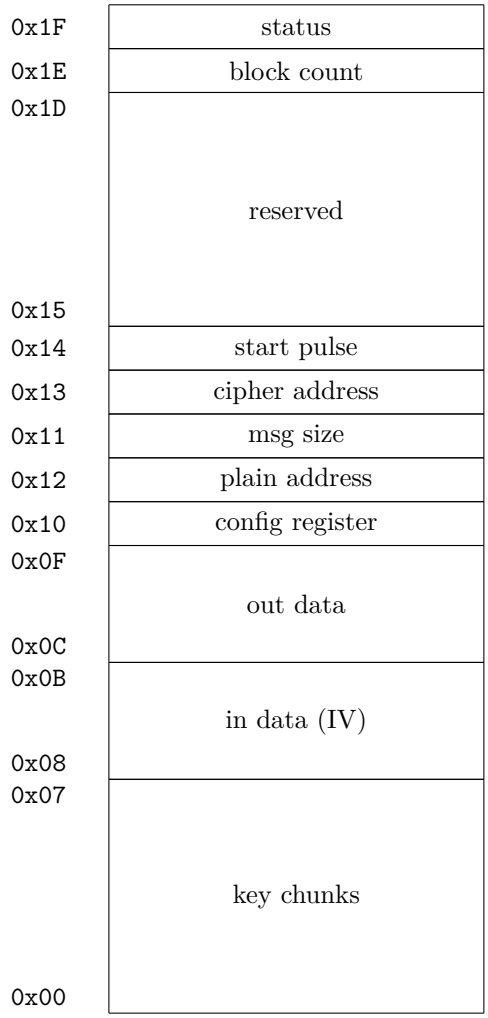


Figure 3.6. Register bank organisation.

address for the plain-text in memory, while the *cipher addr* (0x13) is used to set the cipher-text destination address. These three registers are used by the memory module during an OFB encryption. The start pulse register is a single bit pulse-register used to generate the start condition for an encryption. A write of the value "1" on this register will trigger a pulse of one clock-cycle used as start

signal for the compute module's FSM.

The configuration register is used to set up and prepare the parameters of the computation. In particular, as shown in Figure 3.7, the *config reg* (0x10) it is composed of the following fields:

- [31 : 10] *reserved*
- [9] *mode_ofb* : set to one for OFB encryption, zero for single block encryption.
- [8] *cipher_sel* : zero to select AES, one to select PRESENT.
- [7 : 2] *rounds*: set the number of round for the encryption algorithm.
- [1 : 0] *key_size*: "00" for 80-bit, "01" for 128-bit, "10" for 192-bit, "11" for 256-bit.

All the configuration registers from 0x10 to 0x14 are read-only on the IP side.

The register space from 0x15 to 0x1D is left empty and the register bank will not allow accesses in this region.

The *block cnt* (0x1E) register is read-only for the core and it is constantly updated by the compute unit and holds the number of encrypted blocks.

The last register (0x1F) called *status*, is also read-only for the CPU and holds in the last two bits the status of the compute core and of the memory module FSMs. Specifically the first bit is set when the compute unit is in the idle state, ready to start a new encryption, in the same way the second bit is one when the memory module is ready. If one of this two bits is at zero it means that one of the two module is busy.

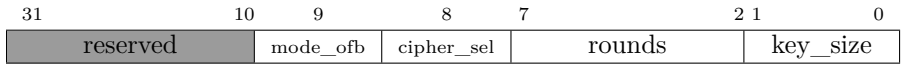


Figure 3.7. *config* register fields.

### 3.2.3 OFB mode: enhancing of the Block-cipher core

The block-cipher core described in section section 3.1 is by itself capable of performing the encryption of a single block of data at a time.

The simplest approach for embedding the block-cipher in a RoCC coprocessor, would be to multiplex the data and key registers and just connect the block-cipher, with the addition of some control logic for correctly timing the input protocol. This approach would work fine for single block encryptions, but it would

incur in the read limitation of the interface when performing a more complex scheme of encryption.

To increase the efficiency of the coprocessor and provide an extra hardware encryption scheme, the block-cipher core is used to compose a more complex computing module able to perform OFB encryptions.

### OFB mode of operation

When the plain-text is larger than the dimension of a single block, several techniques can be used to perform the encryption. The simplest mode of operation is to subdivide the plain-text in $N$ blocks (eventually with some padding in the last block) and perform the simple encryption to each of the blocks. This scheme is called ECB encryption. To avoid security issues, the use of ECB mode should be limited to the plain-texts shorter than a block [5].

Among the mode of operation suggested by NIST [7], there is the OFB mode, which was chosen for being implemented in this design.

The Output Feedback (OFB) is a mode of operation in which the output of the encryption function is xor-ed with the plain text block to produce the cipher-text, at the same time the output of the encryption function is also fed back to become the input for encrypting the next block of plain-text [5]. The algorithm for OFB encryption can be expressed by the following equations:

$$\begin{aligned} O_0 &= IV \\ O_i &= enc_K(O_{i-1}) \\ C_i &= P_i \oplus O_i \end{aligned}$$

Where $C_i$ and $P_i$ are the i-th block of cipher and plain text, $enc_K()$ is a block-cipher encryption function applied with a key $K$, and the *Initialisation Vector (IV)* is usually a random *nonce* (number used only once). A conceptual block scheme of the OFB mode is shown in Figure 3.8. As it can be seen this mode of operation basically creates a stream of pseudo-random numbers, transforming the block-cipher into a stream-cipher.

The OFB mode of operation was chosen because it brings several advantages in the implementation of the coprocessor. In particular for this algorithm the encryption and decryption operations are identical, this means that the decryption just requires exchanging the plain-text with the cipher-text. From an hardware design perspective this means that:

- Only the encryption function is used so there is no need for a decryption core.
- No additional hardware is required for performing the OFB decryption scheme.

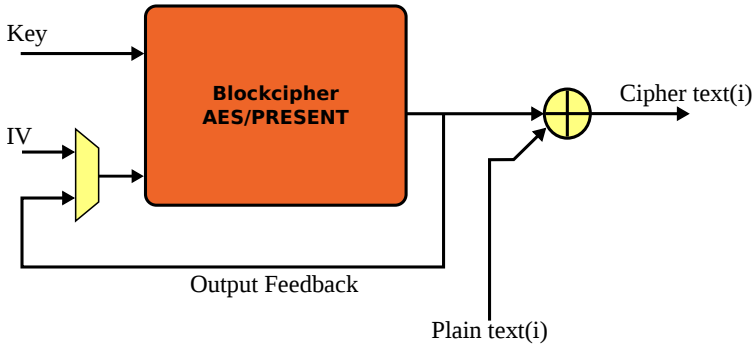


Figure 3.8. OFB algorithm block scheme.

Overall there is no hardware overhead for the support of the OFB decryption. This was ideal since the block-cipher used for this design only implements the encryption operation.

An additional advantage of the OFB encryption scheme is that it implements a stream type of operation and so it is particularly suited for an autonomous implementation.

### Architecture

In order to implement the OFB mode of operation the block-cipher IP is used together with FIFO queues. A FIFO queue is used to implement the feedback and buffer the output block for the next encryption. Another queue is also needed to buffer the encryption output, this one is used to allow decoupling between the stream-generation side and the encryption side. The last FIFO is used to buffer the plain-text data arriving from memory.

The size of the feedback FIFO is fixed to the size of an AES data block, four 32-bit words. While there is no need for a larger queue, this FIFO must be able to store at least a full 128-bit block, to guarantee the correct behaviour of the block-cipher IP and the correct implementation of the input protocol. The output queue on the other hand should be at least twice as big as the feedback FIFO, to guarantee that there is always room for another full block before starting a new encryption. For this reason the size of the output queue was set to be eight 32-bit words (2 blocks), since having a larger one does not give any advantage. The memory FIFO size can be selected arbitrarily, although most optimal value is to have the size of a full 128-bit block. Having a smaller memory queue introduces the latency of the memory accesses in the computation, while having a larger FIFO does not bring any benefit, since the throughput is limited by the latency

of the block-cipher core. In the end the configuration chosen for the FIFOs sizing was the smallest one guaranteeing the maximum throughput.

The computing core takes its input from the *key* and *data in* section of the register bank. In particular the eight key registers are multiplexed into the key data input of the block-cipher core. On the other side the four data-in registers are first multiplexed to select the right data chunk, the result of this selection is then further multiplexed with the output of the feedback queue. This allows to select the source of the data input of the block-cipher IP, this selection is performed by the FSM of the computing core, *CC_FSM* described hereinafter. The computing cores state machine drives also the selection signal for the two data and key multiplexers, controlling the correct sequencing of the input operation.

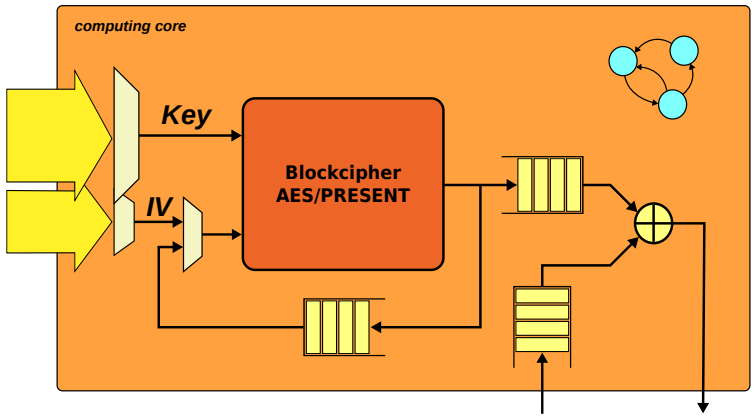


Figure 3.9. OFB module implementation.

The output and memory FIFOs are spilled together to produce the cipher-text chunks, this operation happens only when the two queues have valid data and the memory module is ready. By working in a synchronous manner the two FIFOs are used as a single logical queue.

### 3.2.4 State machine

Overall the FSM controlling the coprocessor is the result of three main state machines working in parallel, *IM_FSM*, *CM_FSM* and *MM_FSM*, as shown in Figure 3.3. The first one, *IM_FSM*, is the FSM controlling the interface module, which allows the *read/write* and *load/store* instruction management. On the other side, directly connected to the register bank there is the compute module, with the main state machine, *CM_FSM*, implementing the logic for the encryption operations. This FSM is activated by a write on the *start* register

of the coprocessor register bank. The third state machine *MM_FSM*, is the one controlling the memory operations during the OFB encryption. This last FSM is decoupled from the main one thanks to the memory and output FIFOs.

The state machine controlling the encryption operation is composed of eleven states, among them there is an initial idle state, eight sequencing states for feeding the key and data chunks to the block-cipher core, a wait state for waiting the result of the encryption and pushing it to the feedback and output queues, and a write-back state for saving the last block produced in the register bank and clear the queues.

As it can be seen in Figure 3.10, the states $key_2$, $key_3$ and $key_5$ perform a check on the length of the key and interrupt the key/data feeding process when the correct number of chunks as been feed. The $key_i$ states implements the block-cipher input protocol, in each of these states the correct key and data registers are selected and the *load_key* and *load_data* signal are correctly driven based on the selected algorithm.

By selecting the OFB mode the state machine will enter in a loop in which, from the *wait* state, when the encryption of a block is finished, it will automatically jump to the $key_0$ state, to immediately start the encryption of a the new block. This procedure will end when the memory module will signal to the FSM the completion of the last plain-text block transfer.

At that point the machine will enter in the *write-back* state (wb) in which the last cipher-text block will be eventually stored in memory and the feedback queue data will be pushed in the *data in* region of the register bank for future encryptions. In this state the FIFOs are checked to be empty, this is mandatory because the two algorithms that are supported (AES and PRESENT) have a different block size, so the coprocessor does not allow to start a new encryption until the queues have been emptied. In this way the content of the queues is cleaned for future encryption, moreover it is possible to start a new OFB encryption from the the last encrypted block, because the feedback is saved in the same registers of the IV. This effectively allows to split an OFB encryption process in different runs of the algorithm.

### 3.2.5 Memory operation

The last main component of the coprocessor is the memory module. This module is the one in charge of filling the memory queue, in the compute module, with the plain-text chunks and to get the produced cipher-text words from the logical output queue and store them in the memory, as shown in Figure 3.3.

This component implements the *mem_req* and *mem_resp* interface and basically performs a round-robin scheme between the load and store requests. To do so it keeps load and store counters which are used both to keep the count of

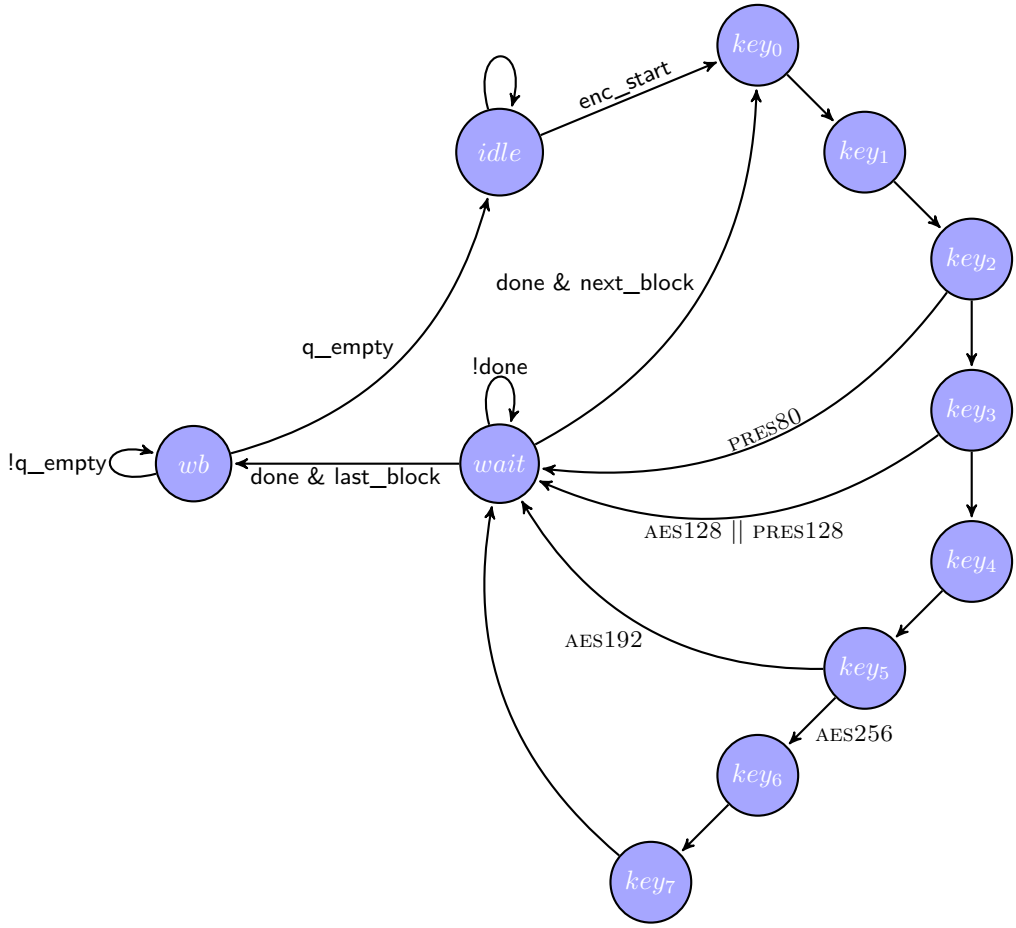


Figure 3.10. State transition diagram of the compute core FSM (CM_FSM).

the memory operations and to compute the addresses for the memory requests, by adding the base address with the counter left shifted by 2, Figure 3.11.

The control logic of this component is mainly implemented by the *MM_FSM*, composed of five states: *idle*, *load*, *store*, *wait_resp* and *wait_queues*.

When the FSM is in the *idle* state the RoCC memory interface is left to the control of the interface module, while the memory module will stay idle until the rise of the start signal. After an OFB encryption is started the FSM goes to the load state in which it will wait for the *mem_req_ready* to be high.

```
base  <= load_base  when (curr_state = s_load) else store_base;
cnt   <= load_cnt_r when (curr_state = s_load) else store_cnt_r;
addr  <= base + (cnt << 2);
```

Figure 3.11. Pseudo code for the address computing logic.

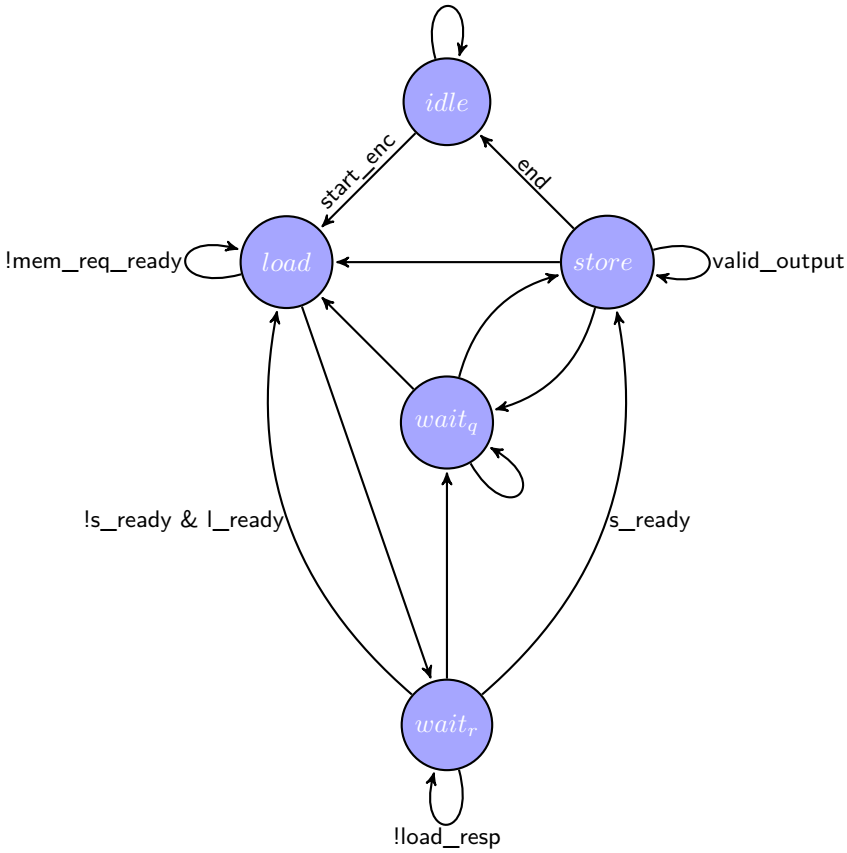


Figure 3.12. State diagram of the memory module controller (MM_FSM). The *wait_resp* state is indicated as *wait_r*.

Once the request has been accepted the FSM goes to the *wait_resp* (wait response) state in which it will wait for the arrival of the memory response for the previous load operation. When the load data arrives it is immediately written

in the memory queue and the state machine decides the next state transition. In particular from the *wait_resp* state the FSM will move to the *store* state when the output FIFO has some new valid data. If this is not the case (the output queue is empty) and the memory queue can receive some data, the FSM will move to the *load* state. When the memory queue is full and output queue is empty, the FSM will go to the *wait_q* state.

The machine will stay in the *wait_q* state untill one of the FIFOs is available, when this condition is satisfied the FSM will jump to either the *load* or *store* state. In case both queues are available at the same time priority is given to the transition in the *store* state. The transition to the *load* state will occur only if there are still load operation to be performed.

The *store* state is the one in which the output queue is emptied and the data is pushed into the memory with a series of store requests. The machine will remain in this state until the output FIFO is empty, once this occurs the machine will go in the *load* state if the memory FIFO is available and there are still loads to be performed. In case the load operation are finished but there are still stores to be performed the FSM will jump to the *wait_q* state. If on the contrary there are no more stores to be performed, the whole procedure is considered finished and the FSM will go back to the *idle* state.

## 3.3 Programming model

This section presents the programming model for the cryptographic RoCC coprocessor. The coprocessor interface module implements the *read/write* and *load/store* instructions defined in the table 2.2, with these instructions it is possible to send and receive data to and from the accelerator. As it is shown in the previous section the register file of the coprocessor is subdivided in different functional registers, each with its own meaning and function.

For programming the coprocessor a sequence of write operation is needed to set up the values of this registers, in particular for performing a single block encryption (ECB mode):

- Write to set-up the *config* register.
- A sequence of writes for the key and data sections.
- A write to the *start* register to begin the computation.

The setup of an OFB encryption requires three extra writes to set-up the plain-text source address, the cipher-text destination address and the size of the message:

- Write the *config* register.

- Write key and IV in the *key* and *data in* registers.
- Write the size of the plain-text to the *msg size* register.
- Write the plain-text source address to the *plain addr* register.
- Write the cipher-text destination address to the *cipher addr* register.
- Write to the *start* register to begin the computation.

These sequences of write instructions can be performed in any order, with the exception of the write in the *start* register which must be performed last.

```
#define ROCC_AES128_ECB_CFG   0b0000100101

int key[] = {...};
int plain[] = {...};
int cipher[4];

void aes128_ecb(int *plain, int *cipher){
    // write configuration
    rocc_write(ROCC_CFG_REG, ROCC_AES128_ECB_CFG);

    rocc_write(ROCC_KEY_REG_BASE + 0, key[0]);
    rocc_write(ROCC_KEY_REG_BASE + 1, key[1]);
    rocc_write(ROCC_KEY_REG_BASE + 2, key[2]);
    rocc_write(ROCC_KEY_REG_BASE + 3, key[3]);

    rocc_write(ROCC_DATA_IN_REG_BASE + 0, plain[0]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 1, plain[1]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 2, plain[2]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 3, plain[3]);

    // Start the encryption
    rocc_write(ROCC_START_REG, 1);

    // save the results
    cipher[0] = rocc_read(ROCC_CIPHER_BASE + 0);
    cipher[1] = rocc_read(ROCC_CIPHER_BASE + 1);
    cipher[2] = rocc_read(ROCC_CIPHER_BASE + 2);
    cipher[3] = rocc_read(ROCC_CIPHER_BASE + 3);
}
```

Figure 3.13. Example of C code for an ECB encryption.

The Figure 3.13 shows a possible code sequence for a function performing the ECB encryption. In the figure the functions *rocc_write(reg, value)* and *rocc_read(reg)* are introduced, these functions can be defined in terms of the actual assembly instructions with the use of *inline assembly* as shown in Figure 3.14.

```
void rocc_write(unsigned int reg, int value){
    asm volatile("rocc_read %[rs1], %[rs2]"
                 :
                 : [rs1]"r"(reg), [rs1]"r"(value));
}

int rocc_read(unsigned int reg){
    int res;
    asm volatile("rocc_read %[rd], %[rs1]"
                 :  [rd]"=r"(res)
                 :  [rs1]"r"(op1));

    return res;
}
```

Figure 3.14. Definition of *rocc_read* and *rocc_write* C functions.

The same sequence of operation can be directly written in *assembly* with few small differences, the main one being the need to manually perform the load and stores for the key, plain and cipher variables. A similar code sequence can be used to write a function launching the OFB encryption, as showed in Figure 3.15.

In the code for the OFB encryption a last read operation is performed on one of the result registers. This read is not really needed since the coprocessor automatically saves the cipher-text in memory, it is instead used as a blocking read to stop the core. The interface module in fact will not accept any read request to the *data out* registers until the encryption is completed. This could be an effective way to stall the core in a low-power scenario.

A non blocking version of the same function can be written by just omitting the last *rocc_read*. In this way the processor would keep executing the code after the function call. As opposed to the previous case, this could be a good strategy for exploiting the parallelism between the CPU and the coprocessor, with the core running its own thread while the accelerator completes the encryption of a message.

```
#define ROCC_AES128_OFB_CFG   0b1000100101

int iv[] = {...};
int key[] = {...};

int plain[] = {...};
int cipher[MSG_SIZE];

int aes128_ofb(int *plain, int *cipher, unsigned size, int *iv){
    // write configuration
    rocc_write(ROCC_CFG_REG, ROCC_AES128_OFB_CFG);

    rocc_write(ROCC_KEY_REG_BASE + 0, key[0]);
    rocc_write(ROCC_KEY_REG_BASE + 1, key[1]);
    rocc_write(ROCC_KEY_REG_BASE + 2, key[2]);
    rocc_write(ROCC_KEY_REG_BASE + 3, key[3]);

    rocc_write(ROCC_DATA_IN_REG_BASE + 0, iv[0]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 1, iv[1]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 2, iv[2]);
    rocc_write(ROCC_DATA_IN_REG_BASE + 3, iv[3]);

    rocc_write(ROCC_MSG_SIZE_REG, size);
    rocc_write(ROCC_PLAIN_ADDR_REG, plain);
    rocc_write(ROCC_CIPHER_ADDR_REG, cipher);

    // Start the encryption
    rocc_write(ROCC_START_REG, 1);

    /** No need to save the result, a blocking read is
      * performed to stop the execution until the
      * encryption is finished
      */
    return rocc_read(ROCC_CIPHER_BASE);
}
```

Figure 3.15. Code sequence for a OFB encryption.

## 3.4 Results and performance analysis

This section reports the results of the simulation of the whole *Rocket SoC* system, focusing on the performance of the coprocessor described in the previous sections. For testing the latency and throughput of the coprocessor a set of test programs were prepared, in which both the ECB and OFB mode were tested.

The algorithm were tested with all the possible key sizes with the set of test vector recommended by NIST in [7]. In the following the results for the AES-128 bit encryption are presented. For these analysis it is considered not only the total number of clock for the operation, but also the total number of instructions. In particular it is also considered the overhead of the normal RISC-V instructions compared to the fraction of custom coprocessor instructions.

### 3.4.1 ECB encryption

The test of the ECB encryption scheme showed all the limitations of the RoCC interface. The test considered a case of single block encryptions placed inside a loop, so that a general figure for an $n$ block encryption could be made.

The block-cipher IP is able to perform an AES-128 encryption in 45 clock cycles. This figure considers the beginning of the encryption from the cycle in which the first data and key chunks are feed, to the cycle in which the last word of the cipher-text is produced. Considering this figure, the proposed design behaves well, taking only 48 clock cycles from the generation of the start pulse (one cycle after the *rocc_write* to the start register) to the return in the idle state. The comparison is showed in Figure 3.16, where the time dime diagram reports the start pulse, state of the computing core and the *data_out* signal of the block-cipher IP.

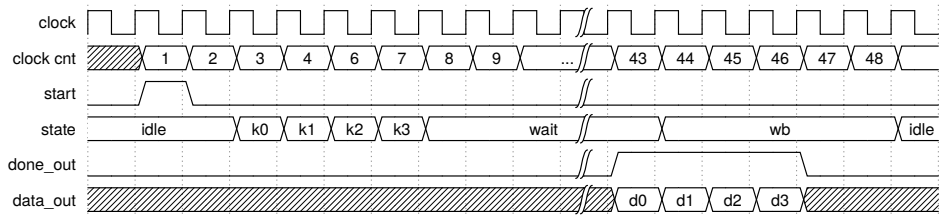


Figure 3.16. Time diagram of the ECB encryption.

The extra 3 clock cycle spent by the coprocessor are due one to the start pulse delay, the other two are caused by the write back of the queues in the register

bank, which add 2 extra clock barriers.

Considering the total execution time and the instruction count things get slightly worse. The test code was compiled in a total of 37 instructions, out of which 14 are coprocessor operations, in particular:

- 1 *rocc_write* for the configuration.
- 4 *rocc_write* for the key.
- 4 *rocc_write* for the data.
- 1 *rocc_write* for the start pulse.
- 4 *rocc_read* for retrieving the cipher-text data.

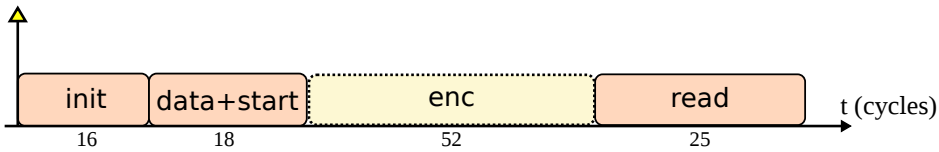


Figure 3.17. Time diagram of the ECB test sequence.

The Figure 3.17 shows a time diagram of the main phases of the program with the relative length in clock cycles. As it can be seen the total time is 111 clock cycles, out of witch just 52 are spent for the encryption (the actual encryption time plus the time for the start writing instruction). The *init* phase consist of the writing of configuration and key, last for 16 clock cycles but it is performed only once. The *data* section last for 18 clock cycles and also accounts for the loop management. In the successive executions the data section can last 13 cycles, since some of the register values are reused (the one for addressing the coprocessor). The *read* section is the one introducing the heaviest overhead with its 25 clock cycles.

Considering a loop for the encryption of $n$ blocks, the total number of clock cycles is computed as: $T_{tot}(n) = 21 + 90n$. This gives a throughput of about 90 clock cycles per block, which is equivalent to 1.42 bits per clock cycle or 142 Mbit/s with a 100 MHz clock. This means that the maximum throughput with this configuration is limited to half the maximum throughput allowed by the block-cipher IP. Being this mode very inefficient, it should not be used if not to perform occasional single block encryptions, the OFB mode, described in the next section, is instead better suited for this coprocessor and should be the preferred mode.

### 3.4.2 OFB encryption

The addition of the OFB mode introduced two main advantages to the coprocessor: the possibility to leave the processor free to execute another task, and the ability to compute the encryptions at the maximum throughput allowed by the block-cipher IP. In particular configuring the coprocessor in the OFB mode allows the encryption of arbitrarily long messages with just a small sequence of initialisation instructions. This could allow the CPU to run without having to continuously exchange data with the accelerator, thus avoiding the overhead introduced by the RoCC interface. Moreover since the coprocessor is not limited by the interactions with the main core, all the encryptions of subsequent blocks are pipelined without introducing any additional delay. This behaviour is shown in Figure 3.18, where it can be seen that the coprocessor can encrypt a $n$ blocks message in a number of clock cycles which is equal to: $T_{coproc}(n) = 1 + 45n + 4$. The proposed design is thus able to achieve the maximum throughput allowed by the block-cipher core, adding only one clock cycle of penalty for the start pulse at the beginning of the encryption, and four clock cycles that are due to the queue management and to the limitation imposed by the memory interface during the store of the last cipher-text block.

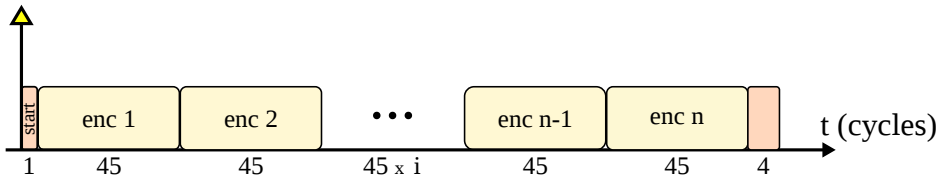


Figure 3.18. Time diagram of the OFB encryption.

When considering the total execution time, the overhead of the instructions and the one caused by the interface, are greatly reduced with respect to the case of the ECB encryption.

The OFB test consisted in a program repeatedly performing the encryption of multi-block messages. This program is meant to test both the case of a normal OFB encryption and the case of an encryption operation divided in multiple runs of the algorithm. The executed test sequence consist of two major section: a set of initialisation instructions to set up the coprocessor configuration, the key and the IV, and the sequence of instructions for setting up the addresses for the encryption and to generate the start command. In particular the *initialisation* phase consist of 20 instructions, of which 9 are coprocessor instructions, the *addr + start* phase is composed of 7 instructions of which four are RoCC instructions.

Out of the 27 total instructions the RoCC ones are:

- 9 *rocc_write* instructions for setting up the configuration, the key and the IV.
- 3 *rocc_write* instructions for the plain and cipher-text size and addresses.
- 1 *rocc_write* instruction for the start pulse.

In Figure 3.19 is reported a time diagram of the different phases, in this case the length of each encryption phase depends on the number of blocks of the plain-text. The initialisation phase takes only 27 clock cycles while the address and start phase cost is eight cycles. This phases might seem to introduce a large overhead, if compared to the 45 clock cycles needed for the encryption of a block, but with this mode this price is payed only once at the beginning.

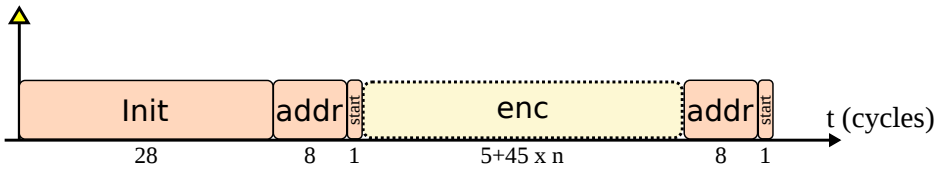


Figure 3.19. Time diagram of the OFB test sequence.

As a result the total time for the encryption of an $n$ block plain-text is roughly computed as $T_{total}(n) = 35 + 5 + 45n$. This figure is much better than the one obtained with the ECB encryption, moreover this method allows also a generally lower instruction count and the possibility to exploit the parallelism with the main core.

For all of these reasons the OFB mode is considered the best way to exploit the proposed architecture.

# Chapter 4

# General coprocessor framework

This chapter outlines some considerations about the design of a coprocessor with the RoCC interface, taking into account the analysis performed in Chapter 2 and making use of the experience coming from the case study presented in Chapter 3. The first section discusses the general architectural choices and concept concerning the development of the accelerators, considering the register file, the use of the interfaces and the state machines for the control of the operations. The second section deals with the software side of the accelerator, considering the possible ways of programming it, the format of the instructions and the addressing modes. In the end the last section provides some considerations about the performance aspect of the coprocessors, like latency and the exploitation of parallelism between the core and the accelerator.

## 4.1 Architecture overview

In general finding generic patterns for coprocessors design is a difficult task, since by definition the coprocessor is a custom special-purpose computing core. This sections tries to generalise the concepts introduced with the design of the block-cipher accelerator, suggesting some design patterns for easing the integration of different IPs without the need to re-design from scratch each new accelerator.

The Figure 4.1 shows the general architectural framework proposed for the design of RoCC coprocessor. As it is shown in the figure, the general coprocessor more or less follows the same architectural pattern described with the

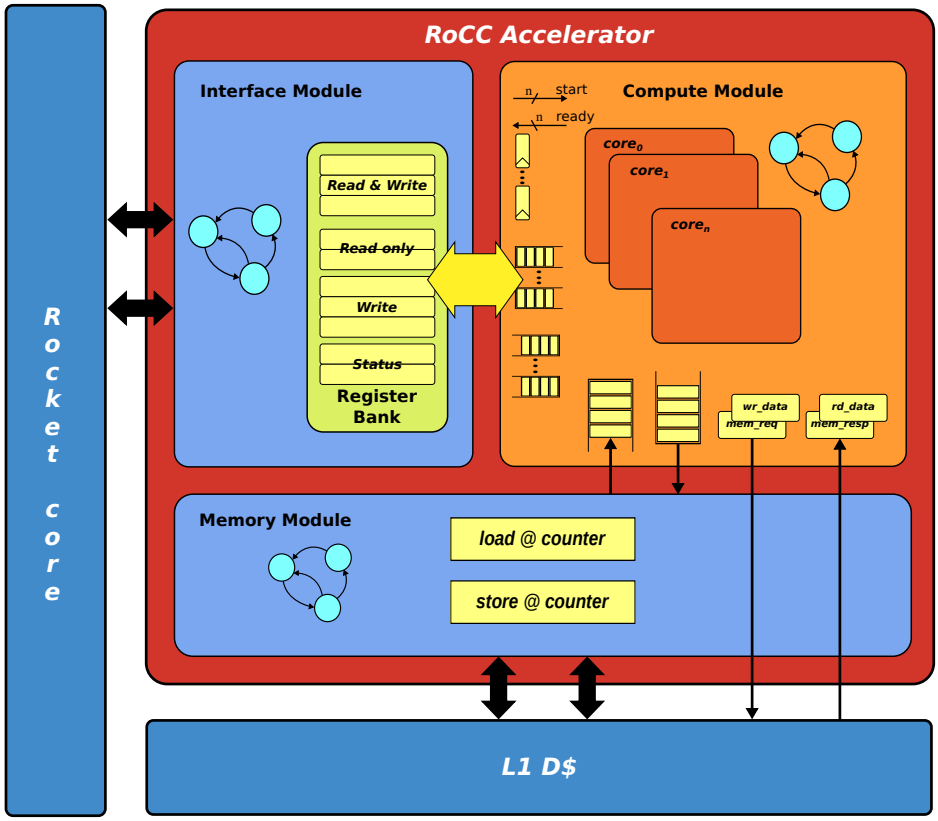


Figure 4.1. General coprocessor architecture.

block-cipher case study. In particular the architecture presents a generic interface module handling the communications with the core and the data transfers with the internal register file. The register bank, as in the case of the block-cipher accelerator, can be spited into sections and will give parallel access to the computing core's side.

In a generic coprocessor, there can be more than one computational units, this units may implement single atomic operations (like a multiply-accumulate) or can be more complex implementing several different functions. Moreover the computing cores may be already existing IPs (like in the case of the block-cipher) or they can be designed on purpose for being part of a RoCC coprocessor. In

general for the first case, it can be convenient to adopt the same approach proposed with the block-cipher case study. With some interface modules dealing with the communication with the core, and some wrapping logic that allows the computing unit to access the register file and maybe talk with the memory.

In case of an on purpose design, the compute unit can be much more integrated and a lot more optimisations can be applied. For example a module specifically designed for being part of a RoCC coprocessor, may directly embed the control logic for performing memory accesses, without the need of an additional module. Moreover the input output protocol of the compute core may be designed to take full advantage of the register bank configuration or of the RoCC *cmd* channel.

### 4.1.1 Interfaces

#### Interface module

As for the block-cipher case study, the proposed architectural framework presents a generic interface module that handle the instructions and the data exchange with CPU. In particular an interface module implementing the *read/write - load/store* instruction model, with a standard interface towards the register bank, can be made agnostic with respect to the type of coprocessor. This can be helpful from a reuse point of view and guarantees the minimal setup for a peripheral-like coprocessor. On the other hand the decode logic and the FSM of the interface module could be extended in order to support different special operations. One example could be the introduction of burst load/store instructions, which can be useful and more efficient for coprocessors operating on a large amount of data. In particular fixed size burst transfers can be implemented to efficiently transfer data with the memory, without suffering from the overhead of programmable burst (wich would require more than one instruction).

Another possible extension to the interface module could be the support for direct responses from the computing unit to the CPU. This would allow the coprocessor to perform a computation and send back the result with just one instruction, without the need of performing multiple read or writes to the register bank.

As an example a vector multiply-accumulate accelerator could make use of both the burst transfer and the bypass of the register file. In particular it would be possible to write a full vector with one instruction, and perform the multiply-accumulate on it directly retrieving the result with a second instruction.

#### Memory interfaces

The direct communications with the L1 data cache are one of the most important features of the RoCC interface. In order to exploit this capability several design

decision can be made. If the target is the use of existing IPs, not specifically designed for being embedded in a RoCC coprocessor, a good choice could be to adopt a strategy similar to the one presented with the block-cipher case study. In particular a coprocessor can make use of a memory module to handle in an efficient way the memory operations, and serving the compute cores with through the use of appropriate interfaces, like the FIFOs in the block-cipher coprocessor. A fixed interface could be adopted, like for example load and store FIFOs and dedicated configuration registers for programming the source and destination addresses. This strategy would allow to design a generic memory module that sequence the load and store operations adopting some kind of scheduling, like for example round robin.

Although this method may seem reasonable, care must be taken to avoid deadlocks. Moreover it is often the case that the specific memory operation performed by a coprocessor follows a regular pattern. In this case it is often easier and more efficient to implement a dedicated state machine to sequence the memory operation, avoiding the probably higher cost of a generic memory module.

Another possible approach could be to devise the computing core embedding the memory control logic. For example for a stream based coprocessor it could be more efficient to directly handle the stream of data from the memory.

However in general the use of a dedicated memory module is suggested, since it makes the overall design more modular and can make easier the design of the compute core. Moreover care must be taken when the memory operation are performed, especially with the load operations, because the memory transactions are not guaranteed to be performed in order. For this reason dividing the duties between a compute core and the memory module seems to be a good design decision.

### Internal interfaces

Inside the coprocessor the movement of data between the different modules can be handled in several ways. In the case of the block-cipher coprocessor the choice was to have the interface module talk with the register bank through the *cfg* interface, while the compute unit have parallel access to all the data and configuration registers. On the memory side the compute unit exchange data through the use of two FIFO interfaces.

This approach can be generalised and extended to the case of many compute units that may talk directly or through the use of queues to both the register bank and the memory. Another possibility could be to have a dual port memory in place of the register file, and handle writing conflicts between the interface module and the compute module.

In general the recommendation is to exploit the register bank with a standard channel on the interface module side, and with full parallel access on the compute IPs side. This allows a basic reuse of modules and it is a reasonably flexible way to provide the compute units with the data they need.

### 4.1.2 Register bank

The register bank is basically the main "interface" of the block-cipher coprocessor at the programming level. Following this principle, the register bank of a generic coprocessor should be designed in order to expose the configuration and the status registers to the CPU, to allow a peripheral like programming of the accelerator. In the proposed architectural framework the register bank is also the main data bridge between the interface module and the computing core.

The register bank of the cryptographic coprocessor has been designed using a code generation tool. Trough this tool it was possible to decide the address map, specify the type of each register (e.g. *read-only*) and the specific fields for the configuration registers.

Regardless of the method used to design it, the register bank should in general provide the following characteristics:

- A read/write port on the interface module side.
- Read/write ports or parallel access on the compute side.
- A protocol to handle conflicting accesses coming from the two sides.
- The possibility of specifying different types of registers.

The read/write port on the interface side is needed to guarantee the reuse and the independence of the interface module from the specific register bank configuration. The protocol to handle the access conflicts between the interface module and the computing core is needed to guarantee data consistency.

The different types of registers are needed to support different configurations, to organise the register bank and to guarantee different properties to the data. As an example the key registers of the block-cipher coprocessor are read-only for the compute module, while they can be both read and written on the interface side. At the same time registers like the *data out* ones are read-only on the interface side and are written by the compute core. Also some special types of registers may be needed, like the *start pulse* register which only generate a pulse and does not hold any value. This types of registers allows to organise and protect the data and to define the read/write accesses rights.

### 4.1.3 State machine

Specifying a generic framework for the design of coprocessor's FSMs is in general very difficult. The state machine model is often used for specifying the control logic for a sequence of operations, and in general, different coprocessors will require different state machines to perform their tasks. However some common considerations can be abstracted from the specific context. In general depending on the complexity of the project two main approaches can be taken for the designg of the control logic.

The first one is to have a single FSM controlling all the operations, from the decode of the commands to the exact sequencing of the compute operations. This approach is feasible for simpler coprocessor, where the compute module does not come with complex input protocols or where no parallelism is required. This method has the advantage of being simple and fast to implement, but often it does not guarantee the best performances, and does not scale well with the complexity of the design.

The other possibility is to split the control logic with the interaction of several state machines. This method may allow higher performance through parallelism, as in the case of the block-cipher coprocessor in which the compute unit performs the encryption while the memory module FSM performs the memory accesses.

The interface module FSM may be adapted to accommodate different functions with the use of wait states like the ones used for waiting load or store memory responses. In this scheme the interface module will enter in one of these states when the start pulse for the specific compute function is triggered. At this point the *ready* signal of the *cmd* sub-interface can be lowered until the end of the execution, when the interface FSM will return to the idle state.

An example of generic FSM for the interface module is shown in Figure 4.2. The different wait states may not be needed if the *ready* signal of the *cmd* sub interface is computed in a combinational way, combining the ready signals coming from different compute modules.

Another common place were FSMs may appear is in the logic for the memory transfers. Since the memory responses are not guaranteed to arrive in-order, a simple strategy may be to enforce the order by imposing waiting states for the responses. This approach is simple to implement, but does not provides the maximum performace. However there are many cases, like for the plain-text loads in the block-cipher coprocessor, in which these wait states does not impose a penalty since the real latency limitation is imposed by the computational unit itself.

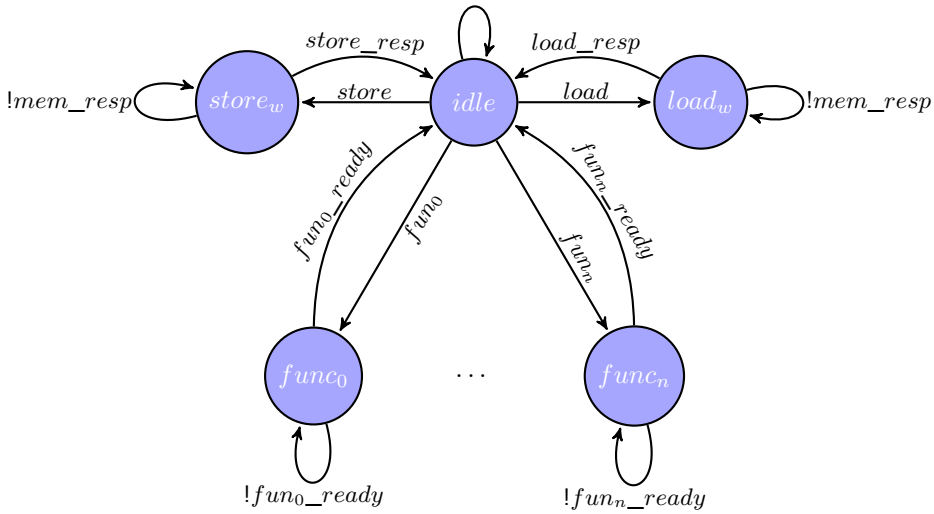


Figure 4.2. Generic interface state machine with $n$ wait states for $n$ different functional modules.

## 4.2 Available instruction

In a generic architectural pattern the instructions to the coprocessor may all be modelled with *read/write* instructions. A generic coprocessor ISA may also benefit from the addition of explicit *load* and *store* instructions, since the RoCC interface provides direct access to the data cache.

As the complexity of the accelerator increases the need for more expressive instructions may arise. In any case the general suggestion is to avoid modifying the available instruction format, since same of the instruction fields influence the behaviour of the core's pipeline. Moreover the 7-bit encoding space should be more than enough to fit the need of complex accelerators.

### 4.2.1 Addressing modes

Another way of organising the coprocessor ISA is to divide it based on the source and destination of each instructions. As in the case of the block-cipher coprocessor the register fields in the instruction can be used to address the coprocessor registers or the CPU registers. Based on this distinctions the instruction set can be expanded to implement other types of instructions. In table 4.1 some of the possible generic instructions, based on the type of addressing are listed. In

| funct7 | xd | xs1 | xs2 | inst_rd | inst_rs1 | inst_rs2 | rs1 | rs2 | operation |
|---|---|---|---|---|---|---|---|---|---|
| **read** | 1 | 0 | 0 | Core | Acc | - | data1 | - | C[inst_rd] ← A[inst_rs1] |
| **read** | 1 | 1 | 0 | Core | Core | - | data1 | - | C[inst_rd] ← A[data1] |
| **write** | 0 | 1 | 0 | Acc | Core | - | data1 | - | A[inst_rd] ← data1 |
| **write** | 0 | 1 | 1 | Acc | Core | Core | data1 | data2 | A[data2] ← data1 |
| **load** | 0 | 1 | 0 | Acc | Core | - | data1 | - | A[inst_rd] ← M[data1] |
| **load** | 0 | 1 | 1 | - | Core | Core | data1 | data2 | A[data2] ← M[data1] |
| **store** | 0 | 1 | 0 | - | Core | Acc | data1 | - | M[data1] ← A[inst_rs2] |
| **store** | 0 | 1 | 1 | - | Core | Core | data1 | data2 | M[data1] ← A[data2] |
| **A_comp** | 0 | 0 | 0 | Acc | Acc | Acc | - | - | A[rd] ← A[inst_rs1] *op* A[inst_rs2] |
| **C_comp** | 0 | 0 | 0 | Core | Core | Core | - | - | C[rd] ← rs1 *op* rs2 |
| **AxC_comp** | 0 | 0 | 0 | Core | Acc | Acc | - | - | C[rd] ← A[inst_rs1] *op* A[inst_rs2] |
| **CxA_comp** | 0 | 1 | 1 | Acc | Core | Core | - | - | A[rd] ← rs1 *op* rs2 |
| **M_comp** | 0 | 1 | 1 | - | Core | Core | src@ | dst@ | M[dst@] ← *op*(M[src@]) |

Table 4.1. Extended coprocessor ISA.

particular an implementation may want to support instructions able to directly perform operations entirely on the integer registers, in those cases an instruction like the *C_comp* may be implemented. On the other hand some coprocessors may benefit from having some instructions able to perform computation entirely inside the accelerator, in those cases the instruction will have the shape of the *A_copm*.

The instructions using the *AxC_comp* or the *CxA_comp* shapes can be modelled with read or write instructions, but a specific design may want to explicitly implement different instructions with these models. In particular different instructions with these format may not access the register bank.

The last proposed model is thought for operations directly working on memory data, this instructions are particularly useful if the accelerator uses data of fixed size, so that another instruction specifying the size of the memory transfer is not needed.

In general the simple *read/write*, *load/store* model should be sufficient to fit the needs of most accelerators, moreover reducing the instruction set makes easier to share the compiler support and the software libraries across different coprocessor. For these reasons the simpler model is considered to be the preferred one.

## 4.3 Performance considerations

This section presents some general considerations on performance taking into account the RoCC interface analysis presented in Chapter 2, and the experience of the block-cipher coprocessor development described in Chapter chapter 3.

From the RoCC analysis and the ECB performance result, it is clear that the RoCC interface imposes a big latency overhead on the data exchange with the CPU.

In particular this latency arises from the position of the interface in the pipeline and from the way the RoCC responses are handled by the core. This limitation is not critical for those coprocessors performing long latency operations and not requiring frequent data exchange with the main core. It is however unadvised to use the RoCC interface to implement tightly coupled coprocessors. In fact this types of accelerators would incur into drastic speed limitations due to the frequent stalls caused by the read operations. As an example if a bit manipulation coprocessor takes just two clock cycles to extract a bit, it will then pay the 5 cycle penalty caused by the stall when the computed value is sent back to the CPU.

For this reason the RoCC interface is considered not suited for tight coupled coprocessor, it is instead better used for the development of decoupled coprocessors, as it is also suggested in the Rocket Chip technical report [2]. In particular the type of interface and the presence of a direct link to the data cache suggests a better use for throughput coprocessors like for cryptography, fir filters or DSP coprocessors in general.

The recommendation is thus to exploit the memory connection for the data stream, and use the read and write operations just for configuration or status information.

# Chapter 5

# Conclusions

This chapter draws the conclusion on the study of the RoCC interface and development of coprocessors for RISC-V architecture. This work was conducted in the LISAN laboratory of CEA LETI of Grenoble and is meant to guide the development of future RoCC coprocessors targeting in particular the L-IoT platform. The objective of this study is to provide an analysis of the RoCC interface and present the development of a cryptographic coprocessor using an available block-cipher core.

The analysis of the Rocket Custom Coprocessor Interface (RoCC) interface is performed with the use of a *read/write* instruction model for the exchange of data between the main core and the coprocessor. The result of the latency study shows that the RoCC interface suffers from big latency overhead mainly due to its late position on the pipeline. This choice was probably made because it guarantees that no exception was thrown by the previous instructions when the RoCC command reaches the coprocessor. Even though dealing with the exception is avoided, postponing the execution of the custom instructions increases the latency of the operations. In particular reading back results from the accelerator imposes a big penalty and often causes stalls on the integer pipeline. In case of reads the latency is made worse also by the way the processor handles the data arriving from the RoCC interface. In fact the writing of a data in the integer register file happens when the write back stage is unoccupied (as in case of branch instructions) or is delayed until that data is needed by another instruction, causing a stall. This result in a big 4 clock cycles penalty when one *rocc_read* is immediately followed by an instruction using that data. When performing more than 4 consecutive *rocc_read* followed by immediate use of the read data the interface start suffering throughput issues. Considering all of this it emerges that the RoCC interface is not the best suited for tight coupled coprocessors and that

it is better to exploit the memory interface when a lot of data must be exchanged.

Following this consideration the proposed design for the cryptographic coprocessor tries to exploit the memory interface to avoid this types of penalties. In particular the design uses the block-cipher IP to implement the OFB mode of operation. The OFB mode has been chosen because it gives several advantages, the main one being that encryption and decryption operations are exactly the same. This means that there is no need for a dedicated decryption module, because only the encryption operation is used, and that the implementation does not require extra hardware for performing the decryption operation. Moreover this mode of operation transforms the block-cipher into a stream-cipher, which is ideal for an autonomous computation. The proposed architecture exploits the memory interface through the use of FIFO queues, by streaming the plain-text blocks and the computed cipher-text blocks. The design reaches the maximum throughput allowed by the block-cipher IP, of one block every 45 clock cycle in AES-128 mode. Moreover once it has been programmed, it is able to autonomously perform a full encryption without relying on the core.

Starting from this design example the guidelines for a generic coprocessor design are outlined. Considering the latency limitations of the interface, the best choice is to exploit the memory channel with coprocessors for throughput applications. All the accelerators that are able to work with streams of data may in fact take advantage of the direct connection with the data cache. On the other hand the applications requiring a tight coupled coprocessor and low latency responses, such as bit manipulation coprocessors, are unsuitable for this interface. For this reasons in a generic framework it is better to consider only throughput applications.

The proposed generic architecture embeds an interface module to handle the basic data transfer between the main core and the coprocessor register file and can be extended to support additional instructions. The register bank implements a common channel on the interface module side, while it can be fully accessed from the computing IPs side. The register file can be divided into sections (read only, read/write, IP writable etc.) and is able to solve access conflict giving the priority to the IP side. A coprocessor can in general embed more than one computing cores, all of this cores will probably have some dedicated configuration registers and some status register. The cores may access the data in different ways, however throughput applications most likely have FIFO like interfaces, and can thus exploit hardware queues for buffering the memory data or the access to the register files. The memory accesses can be handled directly by the cores or can be delegated to a separate module. This memory module can be used to fill and spill some load and store queues, cycling through load and store operations in a round-robin fashion or implementing some kind of priority scheme.

In conclusion this work shows that the core and the discussed interface may

be of interest for future coprocessor development and that a common path for guiding this development can be outlined. Future improvements can take in consideration the other channels of the "extended" RoCC interface, or may try to explore a common ISA for the coprocessor operations so that a common compiler and software framework can be devised.

# Appendix A

# Acronyms

**AES** Advanced Encryption Standard

**AMBA** Advanced Microcontroller Bus Architecture

**APB** Advanced Peripheral Bus

**ASIC** Application-Specific Integrated Circuit

**aUTL** arbitrated Uncached Tile Link

**BAR** Berkeley Architecture Research

**CAD** Computer-Aided Design

**CPU** Central Processing Unit

**CEA** Commissariat à l'énergie atomique et aux énergies alternatives

**EXE** Execute

**ECB** Electronic Code-book

**FIFO** First In First Out

**FSM** Finite State Machine

**FPU** Floating Point Unit

**IF** Instruction Fetch

**ID** Instruction Decode

**IoT** Internet of Things

**IR** Instruction Register

**IT** Information Technology

**IP** Intellectual Property

**ISA** Instruction Set Architecture

**IV** Initialisation Vector

**ECB** Electronic Codebook

**GCC** Gnu Compiler Collection

**LETI** Laboratoire d'électronique et des technologies de l'information

**L-IoT** Low-power Internet of Things

**MEM** Memory

**MMU** Memory Management Unit

**NIST** National Institute of Standards and Technology

**OFB** Output Feedback

**PC** Program Counter

**PTW** Page Table Walker

**RF** Register File

**RISC** Reduced Instruction Set Computer

**RoCC** Rocket Custom Coprocessor Interface

**RTL** Register-Transfer Level

**SoC** System on Chip

**UCB** University of California Berkeley

**WB** Write Back

# Appendix B

# Compiling the tool-chain

This appendix explains the basic steps for the installation, the compilation and the extension of the RISC-V GCC[1] tool-chain.

## B.1 Installing the tool-chain

This section is meant to guide the installation of the RISC-V gnu tool-chain. Most of the steps of the procedure described in this section are also reported in the official documentation of the *Rocket Chip SoC generator* repository[2]. Some of the information can instead be found in the repository for the RISC-V tool-chain, which is embedded in the Rocket Chip repository as a sub-module.

### B.1.1 Download the repository

The first step for setting up the environment is to clone the repository from the official github page: **`https://github.com/freechipsproject/rocket-chip`**, and to download all the sub-modules.

$ *git clone https://github.com/freechipsproject/rocket-chip.git*

$ *cd rocket-chip*

$ *git submodule update –init*

[1] Gnu Compiler Collection (GCC), also indicated as "gcc"

[2] https://github.com/freechipsproject/rocket-chip

### B.1.2 Dependencies

The official documentation in the RISC-V tool-chain repository [3] reports the Ubuntu packages needed for compiling the tool-chain:

$ *sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev*

The Fedora packages are reported as well:

$ *sudo dnf install autoconf automake @development-tools curl dtc libmpc-devel mpfr-devel gmp-devel gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib-devel*

The procedure requires a compiler supporting *C++11* so, if for example GCC is used, a version greater than the 4.8 is needed. It is also possible to use compilers different from the default ones by setting the environment variables **CC** for the C compiler and **CXX** for the C++ compiler. For this work a machine with Red-Hat Scientific Linux release 6.9 (Carbon) was used, *CSH* was used as default shell and gcc-5.1.0 was set as the default compiler:

$ *setenv CC /home/prog/gcc/gcc-5.1.0/bin/gcc*

$ *setenv CXX /home/prog/gcc/gcc-5.1.0/bin/g++*

In order to correctly compile the tool-chain a further step, not explicitly mentioned in the official documentation is needed. In particular the compilation requires the download of some prerequisites. To accomplish this task the *"download_prerequisites"* script must be executed from the "contrib" folder.

$ *cd rocket-chip/riscv-tools/riscv-gnu-toolchain/riscv-gcc/contrib*

$ *./download_prerequisites*

### B.1.3 Install the toolchain

Before starting the build process for the compilation and installation of the tool-chain all the sub-modules must be downloaded. To do so in the following commands mus be executed:

$ *cd rocket-chip/riscv-tools*

[3] https://github.com/riscv/riscv-tools/blob/master/README.md

$ *git submodule update –init –recursive*

Before compiling the risc-v toolchain, an environment variable must be defined, in particular this variable should point to the directory in which the toolchain will be installed.

$ *setenv RISCV /path/to/riscv/toolchain/installation*

If a multi-core machine is used the *MAKEFLAGS* environment variable can be specified to set the number of cores to be used for the compilation:

$ *setenv MAKEFLAGS -jN*

Where $N$ is the number of cores to be used in the compilation process, as an example if four cores are used the command should look like:

$ *setenv MAKEFLAGS -j4*

After all the previous step the compilation can be launched by running the *"build.sh"* script in the *riscv-tools* directory.

$ *cd rocket-chip/riscv-tools/*

$ *./build.sh*

If the 32-bit version of the cross-compiler is needed (for the RV32G ISA) the *"build-rv32ima.sh"* should be executed:

$ *cd rocket-chip/riscv-tools/*

$ *./build-rv32ima.sh*

After the compilation, it can be helpful to extend the **$PATH** variable whit the directory containing the installed tool-chain, specified by the **$RISCV** environment variable.

$ *setenv PATH $PATH:$RISCV/bin*

## B.2 Adding support for custom instructions

This section explains the basic steps that are needed in order to add the support for a *custom* instructions in the RISC-V compiler. Supposing that the instruction that will be added uses all the three integer register fields, as in the case of an *"add rd, rs1, rs2"*, the new instructions will look something like: *"custom rd, rs1, rs2"*. In this example the instruction is going to make use of the *custom-0* opcode.

### B.2.1 Definition of the instruction

The first step to add the instruction is to define the two numeric constants used for masking and matching the instruction. In particular in the file *riscv-binutils-gdb/include/opcode/riscv-opc.h* these couple of constants are defined for every instruction. If the wanted instruction uses one of the custom opcodes then several of this constants are already defined in the file, each using the same opcode and a different configuration for the three bits *xd*, *xs1* and *xs2*. These constants follow the naming convention showed in table B.1. As it can be seen the presence of the suffixes *"_RD"*, *"_RS1"* or *"_RS2"* is conditioned on the value of the respective bit being at one.

| MASK and MATCH | xd | xs1 | xs2 |
|---|---|---|---|
| MASK_CUSTOM0<br>MATCH_CUSTOM0 | 0 | 0 | 0 |
| MASK_CUSTOM0_RD<br>MATCH_CUSTOM0_RD | 1 | 0 | 0 |
| MASK_CUSTOM0_RD_RS1<br>MATCH_CUSTOM0_RD_RS1 | 1 | 1 | 0 |
| MASK_CUSTOM0_RD_RS1_RS2<br>MATCH_CUSTOM0_RD_RS1_RS2 | 1 | 1 | 1 |
| MASK_CUSTOM0_RS1_RS2<br>MATCH_CUSTOM0_RS1_RS2 | 0 | 1 | 1 |
| MASK_CUSTOM0_RS1<br>MATCH_CUSTOM0_RS1 | 0 | 1 | 0 |

Table B.1. Caption

The *"MASK"* constant is used to define bits that are fixed in the instruction, while the *"MATCH"* constant identifies the bit values for the constant bits in the instruction.

```
#define MATCH_CUSTOM0_RD_RS1_RS2 0x700b
#define MASK_CUSTOM0_RD_RS1_RS2 0x707f
```

Figure B.1. Default values for the MATCH and MASK constants.

The figure B.1 shows the default values for the *MASK* and *MATCH* constants for the custom instruction having all the three registers bits at one. In order to fix some other part of the instruction, like for example the *func7* field, the bits

in position 31 to 25 in the *MASK* should be at one, while the same bits in the *MATCH* should hold the decided value for the specified field.

As an example the constant in figure B.2 are used to define the same custom instruction but with the last seven bits fixed at zero as part of the encoding.

```
#define MATCH_CUST 0x700b
#define MASK_CUST  0xfe00707f
```

Figure B.2. Default values for the MATCH and MASK constants.

### B.2.2 Instruction declaration

After the definition of these constants in the same file a macro is used to declare the instructions as showed in figure B.3.

```
DECLARE_INSN(add, MATCH_ADD, MASK_ADD)
DECLARE_INSN(custom, MATCH_CUST, MASK_CUST)
```

Figure B.3. Macro to declare the instruction.

### B.2.3 Update of the instruction's table

Once the previous steps are compleated, the *"riscv_opcodes"* struct in the file *riscv-tools/riscv-gnu-toolchain/riscv-binutils-gdb/opcodes/riscv-opc.c* must be updated with the addition of the new instruction, as shown in figure B.4.

After the opcode data structure has been updated the tool-chain must be re-compiled to apply all the updates.

```
const struct riscv_opcode riscv_opcodes[] =
{
/* name,    isa, operands, match,      mask,      match_func, pinfo.  */
{"add",     "I", "d,s,t",  MATCH_ADD,  MASK_ADD,  match_opcode, 0 },
/*  ...  */
{"custom", "I", "d,s,t",  MATCH_CUST, MASK_CUST, match_opcode, 0 },

/* Terminate the list.  */
{0, 0, 0, 0, 0, 0, 0}
};
```

Figure B.4. RISC-V instruction data structure vector.

# Appendix C

# Generation of the core

This appendix explains the basic steps for configuring the Rocket Core and for generating the RTL verilog description files. Moreover it also presents the custom configuration used to allow the use of an external VHDL module for the RoCC coprocessor.

## C.1 Rocketchip configurations

Configurations are *Scala* classes defining the parameters for the sub-modules available in the Rocket Chip SoC generator. Through the configurations it is possible to customise many aspect of the SoC and of each of its components. As an example it is possible to change the number of sets of the data or instruction cache by modifying the *"nSets"* parameter available for both caches.

The configurations can also be combined with the use of the concatenation operator "++", chaining them starting from the more specific one, to the more general one. As an example to generate a RV32G core the *"DefaultRV32Config"* configuration is used. As it can be seen in figure C.1 this configuration is composed of two configurations, one setting the size of the ISA to 32-bit, the other more general setting the default values for the other parameters.

```
class DefaultRV32Config extends Config(
    new WithRV32 ++ new DefaultConfig
)
```

Figure C.1. Default configuration for a RV32 core. Specified in src/main/scala/rocketchip/Configs.Scala

The figure C.1 shows an exaple of a configuration chainging the "XLen" parameter that is used for setting the feature size of the ISA. This configuration also modifies the FPU and the integer multiplier for adapting them to the 32-bit ISA.

```
class WithRV32 extends Config((site, here, up) => {
  case XLen => 32
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(core = r.core.copy(
      mulDiv = Some(MulDivParams(mulUnroll = 8)),
      fpu = r.core.fpu.map(_.copy(divSqrt = false))))
  }
})
```

Figure C.2. Configuration setting the "XLen" size to 32-bit. From src/main/scala/coreplex/Configs.Scala

### C.1.1 RTL generation

In order to generate the RTL verilog sources the following command must be executed in the *rocket-chip/vsim* folder of the repository:

$ *cd rocket-chip/vsim*

$ *make verilog*

This will generate the core and the SoC with the default configuration. In order to use a different configuration, the "CONFIG" variable must be specified in the command, as an example to generate the verilog for the *"DefaultRV32Config"* configuration the command is:

$ *cd rocket-chip/vsim*

$ *make verilog CONFIG=DefaultRV32Config*

### C.1.2 Generating the RoCC interface

In order to instantiate the RoCC interface with a specified set of accelerators, the default configuration must be extended.

In the *rocket-chip/src/main/scala/coreplex/Configs.Scala* file an example configuration instatiating four coprocessor is present. The configuration, shown in

```
class WithRoccExample extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(rocc = Seq(
      RoCCParams(
        opcodes = OpcodeSet.custom0,
        generator = (p: Parameters) => Module(new AccumulatorExample()(p))),
      RoCCParams(
         opcodes = OpcodeSet.custom1,
         generator = (p: Parameters) => Module(new TranslatorExample()(p)),
         nPTWPorts = 1),
      RoCCParams(
         opcodes = OpcodeSet.custom2,
         generator = (p: Parameters) => Module(new CharacterCountExample()(p)))
    ))
  }
  case RoccMaxTaggedMemXacts => 1
})
```

Figure C.3. WithRoccExample configuration from the "src/main/scala/coreplex/Configs.Scala" file.

figure C.3, is used to map each coprocessor module to one of the custom opcode.

The figure C.4 shows the definition of a new configuration in the *rocketchip/Configs.Scala*, used to apply this configuration in combination with the *DefaultRV32Config*, to obtain a 32-bit core with the RoCC interface and the accelerators.

```
class DefaultRV32Config extends Config(
    new WithRV32 ++ new DefaultConfig)

class RoccExampleRV32Config extends Config(
    new WithRoccExample ++ new DefaultRV32Config)
```

Figure C.4. Definition of a configuration for a 32-bit core with example RoCC accelerators.

## C.2 Custom configuration for generic coprocessors

This section shows the configuration used for the development of the work described in this thesis. In order to connect external modules, not developed in

Chisel, the language provides the possibility of defining "BlackBox" modules, with their own interface. This black-boxes are generated as empty modules only specifying their interface an their connections with the rest of the circuit. It is then enough to specify in a different file a module with the same name and ports and the connection will be recognised by the simulation or synthesis tool.

This feature allows to mix a Chisel design, which is ultimately converted in verilog, with verilog modules or VHDL entities. In order to exploit this possibility, a black-box coprocessor, implementing the RoCC interface, was defined in the file *rocket-chip/src/main/scala/tile/LegacyRoCC.Scala*. The figure C.5 shows the implementation of the classes needed for the black-box coprocessor.

```
abstract class RoCC(implicit p: Parameters) extends CoreModule()(p) {
  val io = new RoCCIO
}

// define a black-box module
class Coprocessor(implicit p: Parameters) extends BlackBox {
  val io = new Bundle{
    val clock = Clock(INPUT)
    val reset = Bool(INPUT)
    val io = new RoCCIO()(p)
  }
  def connect(clk : Clock, rst : Bool, rocc : RoCCIO) = {
    io.clock := clk
    io.reset := rst
    rocc <>  io.io
  }
}

// define a wrapper module that uses the black-box coprocessor
class CoprocessorExample(implicit p: Parameters) extends RoCC()(p){
  val box = Module(new Coprocessor()(p)).connect(clock, reset, io)
}
```

Figure C.5. Definition of the black-box coprocessor.

In order to use the black-box coprocessor, a new configuration is needed instantiating the *"CoporcessorExample"* class module. To do so two different configuration classes are added, one in */src/main/scala/coreplex/Configs.Scala*, used to define the coprocessor map, the other in */src/main/scala/rocketchip/Configs.Scala* to specify a global configuration of the SoC for the use of the black-box coprocessor. The first configuration is shown in figure C.6, while the global configuration

is reported in figure C.7.

```
class WithRoccBlackBox extends Config((site, here, up) => {
  case RocketTilesKey => up(RocketTilesKey, site) map { r =>
    r.copy(rocc = Seq(
      RoCCParams(
        opcodes = OpcodeSet.custom0,
        generator = (p: Parameters) => Module(new CoprocessorExample()(p)))
        )
    )
  }
  case RoccMaxTaggedMemXacts => 1
})
```

Figure C.6. WithRoccBlackBox configuration.

```
class DefaultRV32Config extends Config(
    new WithRV32 ++ new DefaultConfig)

class BlackBoxRoccRV32Config extends Config(
    new WithRoccBlackBox ++ new DefaultRV32Config)
```

Figure C.7. Global configuration for the use of the black-box coprocessor.

After these two configuration are added, a 32-bit Rocket core with an empty RoCC coprocessor module can be generated. To use the "BlackBoxRoccRV32Config" configuration it is sufficient to execute the command:

$ *cd rocket-chip/vsim*

$ *make verilog CONFIG=BlackBoxRoccRV32Config*

# Appendix D

# Code compilation and simulation

This appendix introduces the work-flow used for the compilation of the test programs and its simulations using the Questasim simulator.

## D.1 Compilation

This section briefly presents the work-flow adopted for the compilation of the test program for the Rocket Core. In particular the focus is on a bare-metal execution, without any operating system or any other software layer.

### D.1.1 Entry code

In order to define a basic environment before the execution of the main's program, a small entry procedure, directly written in assembly, is compiled and linked along side the main program. As it is shown in the Figure D.1, the assembly code initialises the machine status register, to enable custom coprocessor instructions, prepares the stack pointer and then performs a jump to the main section of the program.

### D.1.2 Linker script

The entry code initialises the stack pointer considering that the base address for the main memory is mapped in the Rocket Core to the address 0x80000000.

In order to correctly map the entry section and the main program's code in the main memory, a custom linker script is used. The script, reported in Figure

```
#define STACK_POINTER_BASE 0x81000000
#define MSTATUS_XS_MASK    ((1<<15)|(1<<16))

  .section ".text.init", "ax",@progbits
  .globl _start
_start:
  li sp, MSTATUS_XS_MASK
  csrs mstatus, sp

  li sp, STACK_POINTER_BASE

  j main
```

Figure D.1. Entry code for initialising the execution environment.

D.2, places the entry point at the base address for the main memory, while the other code section, data and comment sections are placed at a page of distance (0x1000) from one another.

```
OUTPUT_ARCH( "riscv" )
ENTRY(_start)
SECTIONS
{
  . = 0x80000000;
  .text.init : { *(.text.init) }
  .text ALIGN(0x1000) : { *(.text) }
  .data ALIGN(0x1000) : { *(.data) }
  .comment ALIGN(0x1000) : { *(.comment) }
  .bss : { *(.bss) }
  _end = .;
}
```

Figure D.2. Linker script for the Rocket Core bare metal execution.

### D.1.3 Compile commands

The RISC-V C compiler is used to compile a program for the Rocket Core, within the framework specified by the entry code and the linker script.

Supposing that the program is in a file called *"main.c"*, the entry code is in *"entry.S"*, the linker script file is called *"link.ld"*, and that the target architecture is the RV32G, the command for generating the executable is:

$ *riscv32-unknown-elf-gcc -static -nostdlib -nostartfiles -T link.ld -o prog.elf main.c entry32.S*

This command will generate the "prog.elf" binary file. In order for the program to be used in the simulator, the binary code and data sections must be extracted from the .elf file and converted in a hexadecimal text file. The RISC-V object copy program is used in order to extract the binary information from the prog.elf executable file and store them in the prog.bin file.

$ *riscv32-unknown-elf-objcopy -j .text -j .text.init -j .eh_frame -j .shbss -j .rodata -j .data -j .sdata -O binary prog.elf prog.bin*

After the extraction of the binary information the prog.bin file must be converted in hexadecimal into a text file. To do so the *hexdump* program is used:

$ *hexdump -v -e '1/8 "%08x\n"' prog.bin > prog.txt*

In order to better organise and authomatise the compilation process, the previous commands were organised in a makefile as reported in Figure D.3.

## D.2 Simulation

Once the program is compiled and the executable converted in the hexadecimal text file, the code is ready to be used in the simulator. This section will refer in particular to the Questasim simulator, however the following steps can be adapted more or less to any RTL simulator.

In order to execute the simulation the Verilog RTL code of the Rocket core should be analysed and compiled. In particular after the *make verilog* command the Rocket Chip generator creates two main Verilog files corresponding to the full system on chip and the ram memory behavioral model. These two files will be generated in the *rocket-chip/vsim/generated-src/* folder.

Other than these two main files there are several others Verilog files that implement modules that are needed by the Rocket chip SoC to properly work. These files can be found in the *rocket-chip/vsrc* directory.

To summarise the Verilog files in *rocket-chip/vsim/generated-src/*, for the Rocket chip and for the behavioral ram, the ones in *rocket-chip/vsrc*, plus any

```
BASE = /basedir     # base folder of the code
RISCV = /riscv      # path of the riscv tool-chain

PROG = prog.elf     # name of the executable
BINARY = prog.bin   # name of the binary file
MEM = prog.txt      # name of the memory (hexadecimal) file
DUMP = prog.dupm    # name of the dump file
CSRC = main.c       # .c and .h source files

ARCH = 32
ARCH_PREFIX = riscv$(ARCH)-unknown-elf

CC = $(RISCV)/bin/$(ARCH_PREFIX)-gcc           # compiler
OBJCP = $(RISCV)/bin/$(ARCH_PREFIX)-objcopy    # object copy
OBJDUMP = $(RISCV)/bin/$(ARCH_PREFIX)-objdump  # object dump
HEXDUMP = hexdump                              # hex converter

LINK_FILE = $(BASE)/link.ld
IDIR = $(BASE)/include          # directories for .h files

CC_FLAGS += -static -I$(IDIR) -nostdlib -nostartfiles -T $(LINK_FILE)
OBJCP_FLAGS += -j .text -j .text.init -j .eh_frame -j .shbss \
               -j .rodata -j .data -j .sdata -O binary
OBJDUMP_FLAGS += -D
HEXDUMP_FLAGS += -v -e '1/8 "%08x\n"'

all: $(PROG) $(BINARY) $(MEM) $(DUMP)
$(PROG) : $(CSRC) $(LINK_FILE)
        $(CC) $(CC_FLAGS) -o $(PROG) $(CSRC) $(BASE)/include/entry_$(ARCH).S
$(BINARY) : $(PROG)
        $(OBJCP) $(OBJCP_FLAGS) $(PROG) $(BINARY)
$(MEM): $(BINARY)
        $(HEXDUMP) $(HEXDUMP_FLAGS) $(BINARY) > $(MEM)
$(DUMP) : $(PROG)
        $(OBJDUMP) $(OBJDUMP_FLAGS) $(PROG) > $(DUMP)

.PHONY: clean
clean:
        rm $(PROG) $(BINARY) $(MEM) $(DUMP)
```

Figure D.3. Makefile for the compilation and conversion of RISC-V programs.

other RTL file, like for example the VHDL files for the coprocessor described in this work, should be imported, analysed and compiled inside the simulator program.

Once the compilation is done the hexadecimal file obtained by the compilation of the code should be loaded in the main memory. The memory in wich to load

the program can be identyfied in the memory list view of the simulator, because it should be largest memory. In order to automatise this process the code in Figure D.4 can be placed in an *load_memory.do* script and executed from the simulator console with the command *do load_memory.do*.

```
mem load -i prog.txt -format hex \
    /TestDriver/testHarness/SimAXIMem/AXI4RAM/mem/mem_ext/ram
```

Figure D.4. Load memory command.

# Bibliography


[1] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[3] D. H. Bui, D. Puschini, S. Bacles-Min, E. Beigné, and X. T. Tran. Ultra low-power and low-energy 32-bit datapath aes architecture for iot applications. In *2016 International Conference on IC Design and Technology (ICICDT)*, pages 1–4, June 2016.

[4] D. H. Bui, D. Puschini, S. Bacles-Min, E. Beigné, and X. T. Tran. Aes datapath optimization strategies for low-power low-energy multisecurity-level internet-of-thing applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–10, 2017.

[5] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education, 2016.

[6] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. *PRESENT: An Ultra-Lightweight Block Cipher*, pages 450–466. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[7] Morris J. Dworkin. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2001.